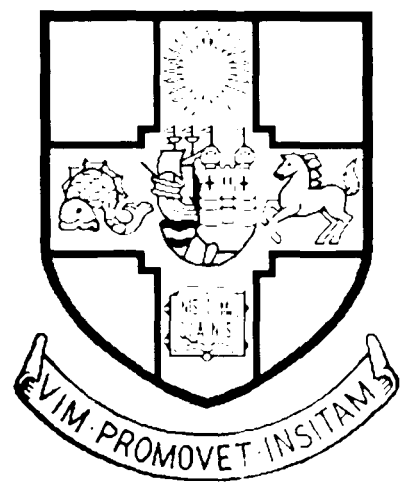
University of Bristol

DEPARTMENT OF AEROSPACE ENGINEERING




# RECONFIGURABLE INTEGRATED MODULAR AVIONICS

**Tomasz Adam Omiecinski**

A thesis submitted to the University of Bristol for the degree of Doctor of Philosophy
in the Faculty of Engineering

December 1998

This thesis contains approximately 76,000 words

# Abstract

Integrated Modular Avionics standardises hardware and software platforms of Line Replaceable Modules (LRMs) and other system components in order to reduce the overall cost of system development, operation and maintenance. Several identical processing units within a cabinet, and fast communication media in the form of a backplane bus introduces further possibility of reconfiguring the system in terms of changing the applications performed by particular core LRMs.

In this thesis a study into Reconfigurable Integrated Modular Avionics is presented. The main objectives of the project were to investigate the benefits, and feasibility of, employing autonomous dynamic in-flight reconfiguration of the system as a means for providing fault-tolerance. In this approach, allowing processing modules to change their function permits the system to share the redundant modules as well as sacrificing less important avionics functions to sustain the more critical applications.

Various architecture examples are reviewed in order to establish a system design that would support reconfiguration at a minimal cost. Two modified ARINC 651 architecture examples are proposed for implementation of dynamic in-flight reconfiguration. The benefits of reconfiguration are identified with the use of Markov state space analysis, and are found to be substantial with respect to the reduced number of redundant processing modules required to implement the system functions within the safety requirements.

Suitable reconfiguration schemes are identified, and the most promising one is formally specified with the use of the Vienna Development Method. The safety properties of the scheme are shown based on the specification. In order to study the feasibility of autonomous dynamic reconfiguration, the scheme is implemented into two distinct systems, and the results of the practical observation of the system behaviour are presented and discussed.

As the project was sponsored by the UK Civil Aviation Authority, a number of certification issues related to reconfigurable avionics systems are identified and discussed based on the practical implementation and previous theoretical analysis.

It is concluded that dynamic in-flight reconfiguration of avionics systems can lead to substantial savings in terms of the reduced number of required core LRMs, and greater fault-tolerance than traditional non-reconfigurable systems.

# Dedication and acknowledgements.

I would like to thank Mr David Johnson who proposed this research and was my initial supervisor and mentor. I would also like to thank Professor Martin Lowson for taking the supervising responsibility after David's departure from the University.

This research would not be possible without the financial support of the UK Civil Aviation Authority to whom I am gratefully indebted. I would like to specifically thank the following people from CAA for their enthusiastic involvement throughout the project: Mr Steve Griffin, Mr Dan Hawkes and Ms Pippa Moore.

Special thanks go to British Aerospace at Filton, Bristol, who presented me with the opportunity for implementing the system with the use of their Systems Digital Control Laboratory. In particular Mr Gary Wicks, Mr John Rice and Mr Gary Yelland deserve mention for their technical expertise and help.

Finally, there is this crazy bunch of friends who made this research and life in general far more interesting and enjoyable than I could have expected, so I would like to thank them all.

I would like to dedicate this work to my parents who patiently waited for me in Poland for all this time.

# Declaration

I declare that the work in this dissertation was carried out in accordance with the Regulations of the University of Bristol. The work is original except where indicated by special reference in the text, and no part of the dissertation has been submitted for any other degree.

Any views expressed in the dissertation are those of the author and in no way represent those of the University of Bristol.

The dissertation has not been presented to any other University for examination either in the United Kingdom or overseas.

SIGNED: DATE:

# Table of Contents

# Index of Tables

# Index of Figures

# Index of Code Examples

# Chapter 1. Introduction

## 1.1. Introduction

This paper refers to research into the area of Reconfigurable Integrated Modular Avionics (RIMA), which was conducted between September 1995 and October 1998 at the University of Bristol, England. The original proposal came from Mr David Johnson, at the time a lecturer in the Department of Aerospace Engineering, and it was the UK Civil Aviation Authority who have taken interest in the subject and have decided to sponsor the research. Mr Steve Griffin was assigned the project manager to represent the interests of the UK CAA, Mr Dan Hawkes and Ms Pippa Moore also provided their invaluable comments and suggestions throughout the project.

On the University side the project was initially overseen by Mr David Johnson, and after his departure in August '97, by Prof. Martin Lowson, the Head of Department of Aerospace Engineering at the University of Bristol.

The main aim of the research was to investigate the feasibility and possible benefits of employment of dynamic in-flight reconfiguration of an integrated avionics system, both in terms of cost reduction and improved system safety. Further objectives included:

- review of current IMA standards and proposals, and identification of a possible reconfigurable IMA architecture,
- analysis of requirements with respect to system configuration, its safety and dispatch availability,
- analysis of requirements, proposal and formal definition of a reconfiguration method suitable for commercial applications,
- proof of the reconfiguration method integrity,
- demonstration of dynamic reconfiguration,
- establishment of certification issues related to the domain of reconfigurable avionics systems.

The work was planned to be completed within three years between October '95 and October '98.

At different stages various organisations and companies were involved in the programme. This includes but is not limited to British Aerospace AIRBUS Ltd., GEC Marconi, Smiths Industries and British Airways. Close links between the University and British Aerospace allowed some of the work to be conducted with the use of the System Digital Control Laboratory, which involved Mr Dave Cole and Mr Gary Wicks in the initial phases of the programme, and later Mr John Rice and Mr Gary Yelland who provided technical help.

## 1.2. Integrated Modular Avionics

The total cost of avionics systems in terms of their development, manufacturing, spares, maintenance, etc. constitutes a significant factor in the cost of a commercial aircraft, and it has been commonly estimated as approximately some 30% - 40% of the total aircraft cost. Thus, there is a significant pressure to identify an alternative to the traditional Line Replaceable Module (LRU) or black box based designs, that would allow a reduction of overall cost and yet they would maintain or improve the system safety. This becomes particularly important in the situation where air traffic is continually increasing, and very high capacity passenger aircraft are likely to be introduced in the near future.

Integrated Modular Avionics (IMA) essentially proposes the replacement of many different types of LRUs, by a few types of standardised Line Replaceable Modules (LRMs), that are to be mounted in a number of racks or cabinets. Modules mounted in the same rack are expected to communicate with one another via a cabinet dedicated backplane bus, whilst the communications with other parts of the aircraft systems should be carried by separate system data buses (most probably complying with the ARINC 629 standard [1]). The standards for commercial IMA are defined in ARINC 651 [2], which also proposes various architecture examples for integrated avionics which are later reviewed in Chapter 2.

The commonality of hardware and software components is expected to allow for reduction of the initial development cost, as well as eliminating the necessity for storage of many different types of spare parts, and it should simplify the maintenance procedures. Moreover, as the LRMs become standardised, the previously inherent dedication between the control unit and its avionics function disappears, which indicates a possibility for re-allocating applications throughout the system. Allowing the system to

dynamically reconfigure could mean that instead of carrying a great number of dedicated backup units, the required system safety [3] could be maintained by sharing the backup modules between many LRMs, or even by the introduction of backup units performing non-important functions. It is expected that the full potential of IMA systems can only be exploited if the system is allowed dynamic reconfiguration during the flight.

## 1.3. Reconfigurable Integrated Modular Avionics

The idea of Reconfigurable Integrated Modular Avionics (RIMA) is strongly based on the principles of IMA. However, unlike in traditional non-reconfigurable systems where functions are statically bound to processing units, the applications can be re-allocated between different LRMs.

In systems employing dynamic reconfiguration a processing module can be required to perform one of many different functions depending on the current state of the system. In the event of a module failure, a RIMA system tries to minimise the safety hazard by choosing one of the less critical modules to take over the lost function. As the dedication between units and functions no longer exists, it is expected that RIMA systems should be able to operate with greatly reduced processing module redundancy (backup LRMs can be shared by multiple functions). It is also anticipated that other cost related benefits such as reduced weight of the system and its lower power consumption will follow the implementation of reconfigurable avionics systems.

As it is the reconfiguration process itself that attempts to minimise the safety hazard, it has to be considered safety related or even safety-critical, and as such it has to comply with requirements for aircraft systems [3]. Any design or implementation errors contained within the reconfiguration algorithm can easily cause undesirable interference between avionics systems, or even lead to catastrophic failure conditions (e.g. due to an implementation error a critical function could reconfigure to a less important one). Therefore, features such as the reconfiguration scheme determinism and integrity are of great importance in RIMA. Also, in order to eliminate possible single points of failure, the reconfiguration process should not rely on a single controlling module or device. Instead, the reconfiguration process

should be distributed, with each module able to act autonomously based on its observations of the system state.

When considering Reconfigurable IMA as a descendant of IMA some generic issues related to the reconfiguration method need to be discussed, that refer to the extent that reconfiguration is to be exploited by the system. In the simplest example of RIMA reconfiguration would only be used when the aircraft is grounded to re-utilise the available resources. Although this approach is likely to be simplest and easiest to introduce, it does not take the full advantage a reconfigurable system can provide (during the flight the avionics system is still essentially non-reconfigurable). Therefore, to fully investigate the benefits of RIMA one has to look at dynamic in-flight reconfiguration of the system in either global (across the whole system), local (within a cabinet only) or a combined manner.

### 1.3.1. Globally reconfigurable IMA

In global reconfiguration every module can at any time encounter the necessity to perform any of the system functions. A task of any core LRM can be assigned to any other core LRM, thus the function migration process involves the whole system. Clearly efforts have to be made towards the reduction of the fetch distance (undertaking the missing function by the nearest suitable module and obtaining the software from the closest possible source), but even then, in the case of very many module failures, there may be a need for downloading the software for a relevant application from distant sources.

Reconfiguration across the whole system has the potential for providing a much higher level of fault-tolerance than the local approach. Assuming that 50% of functions have to be performed (the flight critical functions), the global reconfiguration scheme would allow 50% of core modules (processing units) to fail without violating the safety constraints.[1] Similarly the system can lose whole cabinets due to power supply fault or backplane faults and the rest of the system will still preserve the essential functions.

Some problems with ARINC 629 may arise, when it has to be used as the communication medium between cabinets. If the application software has to be fetched from a remote module in a different

cabinet or from an external Software Store (SS), the data traffic on the system bus would increase in a major way, thus interfering with the whole system. Also, the time required to download the software may prove critical, as downloading 1 MB of software on a totally free ARINC 629 data bus would take approximately $8\text{Mb} / (2\text{Mb} / \text{s}) \approx 4\,\text{sec}$.

Since the use of ARINC 629 as a software downloading bus is not advisable, local task storing has to be taken into consideration. In this approach every module stores the software for all tasks that it may ever be required to perform, in its local memory. To prevent a loss of an application due to a transient power supply fault (e.g. temporary loss of power), some non-volatile memory would have to be used for that purpose.

Simple calculations for the AIRBUS A320 avionics implemented as global RIMA indicate the need for about 90 MB of non-volatile memory mounted into each processing module (every module stores all 88 applications of 1 MB each[2]). The amount of 90 MB of non-volatile memory installed in every single LRM seems to be highly unfeasible (currently, the highest density Intel® StrataFlash™ memory chips could store up to 8 MB of data, thus to achieve the necessary capacity one would require some eleven chips on board). Moreover, if some new, more complex avionics systems would have to be considered, the memory requirements would grow even more.

Introducing a dedicated redundant SS to every cabinet and providing a fast software bus (SB) could solve the problem of reconfiguration delay by reducing the time required for software downloading, and at the same time it would eliminate the need for an unfeasible amount of non-volatile memory to be installed on each core LRM. This, however, would increase the cost and complexity of the system, and questions relating to SS becoming the cabinet single point of failure and thus relating to the integrity of the method would still have to be asked.

---

[1]Such a level of fault tolerance may not be necessary if the probability of an event (or a sequence of events) leading to such a condition is sufficiently low.

[2] As the actual size of each application remains a commercial secret of British Aerospace AIRBUS Ltd., the size has been assumed to be of the order of 1 MB, which should be sufficiently large to accommodate for the growth in future avionics systems.

Finally, it is extremely likely that the global RIMA systems will be highly complex and unsuitable for formal proof methods. Similarly the testing and verification of such systems become very time and resource consuming and their management intensely difficult, thus rendering the global reconfiguration approach rather unattractive for safety-critical applications.

### 1.3.2. Locally reconfigurable IMA

Local reconfiguration schemes require that the avionics functions will only migrate within a cabinet, thus any module in a cabinet can only be expected to undertake one of the functions that were assigned to the cabinet during system integration. The fetch distance is obviously limited to the cabinet or an external SS, and as such it should be on average shorter than in the case of global reconfiguration schemes.

The local reconfiguration approach has not got the flexibility of the global method. With the assumption that 50% of functions in a cabinet are required by aircraft safety, one can easily realise that now the loss of anything above 50% of the modules in a cabinet may lead to the loss of a critical function. This means, that in a system implementing 88 functions (e.g. A320) with ten core LRMs per cabinet, the loss of just six modules may lead to hazardous or critical failure modes, while in the case of global reconfiguration the number would be around forty five. Since, however, the loss of that many modules in a single cabinet is expected to be extremely improbable[3], such level of system fault tolerance should be quite acceptable.

Local reconfiguration schemes should also address the problem of choosing a suitable function subset to be run by a cabinet. In order to achieve high reconfigurability, any set should contain as few critical functions as possible, and preferably spread the function criticalities equally across the whole range to allow for the loss of non-important functions in order to sustain the more critical ones.

Unlike with the global reconfiguration method, the local approach does not require a huge amount of memory per module if the necessary tasks are to be stored locally. Assuming again the configuration of ten modules per cabinet (ten applications of 1 MB each), every module requires only about 10 MB of on-

[3]Clearly, there is a possibility of losing even a greater number of core modules in a cabinet as a consequence of some external conditions (e.g. fire). That should, however, be extremely improbable.

board non-volatile memory. The feasibility of such an approach is much greater than in the case of global reconfiguration schemes.

Moreover, some reconfiguration algorithms may also employ the backplane bus for software fetching from a local source. For example, each module could store just a number of programs, and thus the redundant copies of all applications could be downloaded via the backplane bus by any module required to reconfigure. This method, although possibly slower in some cases (i.e. should the software be available locally no speed difference would be noticeable), needs just a few megabytes of non-volatile memory per module.

Local reconfiguration schemes do not introduce any additional traffic on the ARINC 629 system bus, and they avoid the possible bottleneck by using the ARINC 659 backplane bus for inter-module communications. However, even simple calculations show that with a totally free ARINC 659 bus, the delay related to downloading of a 1 MB application would be of the order of 250-300 ms[4], which may be unacceptable for many functions. Thus either a module has to store all functions in its memory, or an additional very high-throughput bus will be required for software downloading purposes, which would clearly increase the cost and complexity of the system.

Generally, systems based on the local reconfiguration approach seem to be potentially simpler, and thus easier to prove, verify and certify. Also, although lower than in the case of global schemes, the level of reconfigurability should be sufficiently high to meet all safety requirements [3]. Taking into account the discussion from section 1.3.1 and 1.3.2 it has been decided that the research should focus on locally reconfigurable IMA systems.

## 1.4. Organisation of the remaining part of the paper

In Chapter 2 the IMA architecture examples as seen in ARINC 651 [2] are reviewed with respect to their applicability to RIMA, based on which two possible RIMA designs are proposed. The following chapter

[4]With the nominal bus throughput of 30 Mb/s, uploading the 1 MB (8 Mb) of software would introduce a delay of: 8Mb / (30Mb / s) = 0.26s.

reviews various reconfiguration methods as implemented in different applications of distributed computing systems. In Chapter 4 the optimal configuration of a RIMA system with respect to processing unit redundancy, system safety and system dispatch availability is established and analysed. Chapter 5 identifies requirements that reconfiguration schemes employed in RIMA systems should conform to, and in Chapter 6 various classes of such schemes are presented and discussed. A selected reconfiguration scheme is further formally specified in Chapter 7, and its various properties are proven and discussed in Chapter 8. In the following chapter two different systems that were designed to implement dynamic reconfiguration and were subsequently used for its demonstration are described, and various practical issues regarding such systems are also discussed. Finally, in Chapter 10, certification issues related to Reconfigurable Integrated Modular Avionics are identified and discussed, and recommendations for future work are given in Chapter 11.

# Chapter 2. Architecture

## 2.1. Introduction

In view of the close correspondence between IMA and RIMA systems, there is a strong incentive to base reconfigurable systems on architecture developed for traditional non-reconfigurable systems. Clearly, adding the reconfiguration scheme "on top" of an already existing IMA system would allow great reduction in the cost of initial system design, as well as simplifying the certification procedures of the final architecture.

Five different IMA architecture examples can be found in the ARINC 651 report. It is believed that those design proposals came from various companies involved in research into integrated avionics systems, and as such they should be considered a particularly attractive choice for the implementation of a reconfigurable system. This simply follows the fact that all the initial design work has already been completed, and in addition the organisations in question should be less reluctant to contemplate the introduction of dynamic reconfiguration into their system, if it does not require a great deal of additional effort. Thus, the IMA architecture examples have been chosen for investigation in preference to those found in [4], [5], [6], [7] or [8], and referring to various parallel or distributed computing systems.

In this chapter IMA architecture examples as seen in the ARINC 651 report are reviewed with respect to their possible application to a reconfigurable system. Two RIMA architecture examples are then proposed, which aim to maximise the system reconfigurability and related benefits. These examples also attempt to minimise changes required to the underlying IMA design to avoid unnecessary development costs.

## 2.2. Review of ARINC 651 IMA architecture proposals

In this section IMA architecture examples from ARINC 651 (section 6) are analysed and their strengths and weaknesses are identified. Also, their applicability for reconfiguration is investigated and assessed.

## 2.2.1. ARINC 651 architecture "A"

### 2.2.1.1. Overview

Architecture "A" follows the ARINC 700 based avionics systems design and consists of four cabinets interconnected via the ARINC 629 data bus (see Figure 2.1).

Every core LRM implements a particular avionics function that resides in the module non-volatile memory and is performed by a central processor with the use of the module Random Access Memory (RAM). Every processing module executes a single application, so there is no need for any sophisticated memory management as function partitioning is provided on the module level. All modules (processing, I/O, gateways/bus bridges) include an interface to the backplane bus - used for the inter-module communications - and to the power supply.

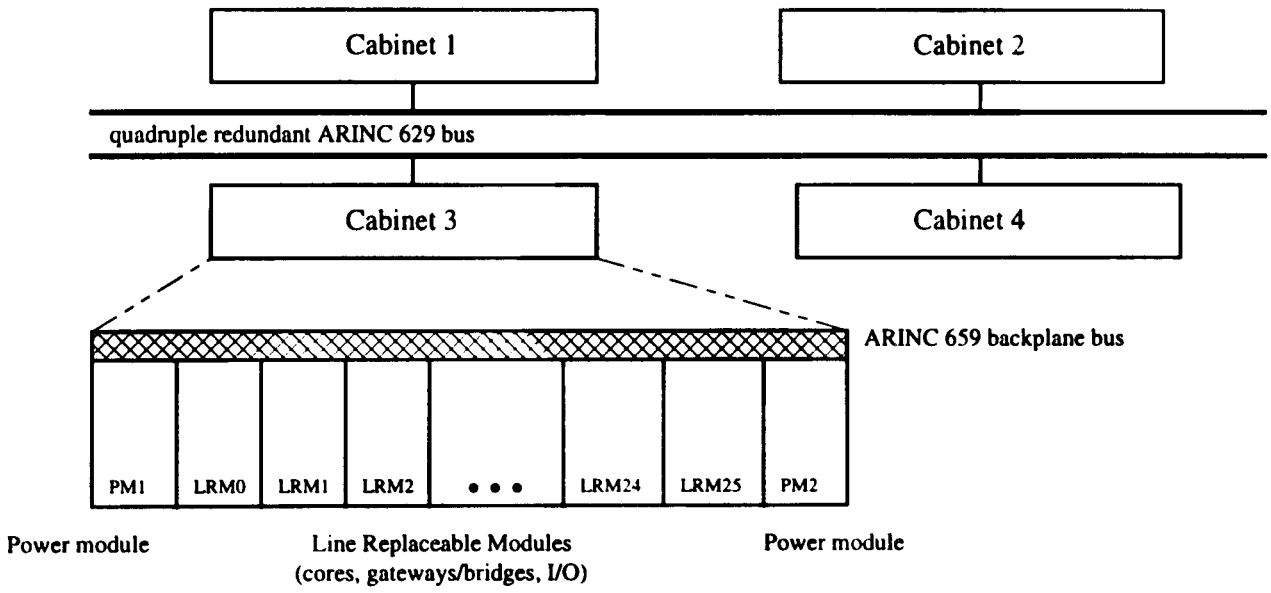



**Figure 2.1. ARINC 651 IMA architecture example "A".**

Although each core LRM performs strictly one program it can use multiple processors internally in order to provide a sufficient level of fault tolerance. Therefore the processing element in the module may be of the form of an n-version multiprocessor (where each processor performs identical functions and the results are voted), or a single version multiprocessor (where each processor in the processing element may perform different sub-functions implemented by the software).

The design of a cabinet employs primarily similar hardware redundancy in order to provide fault tolerance. However, the use of dissimilar hardware is not excluded if there is a need for such redundancy.

Since modules in cabinets are closely bound to specific avionics functions, different cabinets may be of a different design, for example with respect to the presence of dedicated I/O modules. Gateway or bus bridge modules are responsible for data exchange between the ARINC 629 global data bus and the ARINC 659 backplane bus; at the same time I/O modules convert analogue signals from simple devices (which are unable to connect directly to the ARINC 629 data bus) to digital data and make it available on the backplane bus.

#### 2.2.1.2. Advantages and shortcomings

In this section various features of the architecture being discussed are presented with emphasis on their beneficial characteristics or associated drawbacks.

Advantages:

- Fault detection and isolation is restricted to a single module and thus could prove to be relatively simple. Good function partitioning follows naturally from the design as exactly one program is executed by any single module.
- There is no need for any sophisticated memory management to provide software partitioning in a module, since each processing LRM memory is dedicated to a single program. Some memory management may, however, be necessary if multiple processes are needed to implement a single program on a single processing element.
- A failure of any core LRM implies a loss of a single function, so that error messages for the crew can be simple and unambiguous.

Shortcomings:

- A decrease in flexibility of avionics systems employing architecture "A" can be expected, based on a somewhat artificial assumption that there are exactly four cabinets. Each cabinet would provide the processing power for a quarter of the system functions, so the possibilities of a "free" assignment of different functions to different cabinets would be significantly reduced. Similarly in case of future growth in avionics, the explicit number of four cabinets per system may be too restrictive and installing a huge number of modules to a cabinet may not be feasible. However, this IMA architecture could be employed if the restriction on the number of cabinets were abandoned.

- All cabinets are specialised to perform certain avionics functions based on the availability of specialised I/O LRMs, and thus their interchangeability is rather restricted. However, since the idea of IMA implies only module standardisation, cabinet design dissimilarity does not violate any architectural constraints.
- Some bottlenecks can be encountered on gateways and/or bus bridges, especially as ARINC 629 bus is about fifteen times as slow as ARINC 659. However, since IMA architecture is based on those two buses, the problem of throughput difference is rather generic and possible delays may occur even when a module communicates with the direct use of the global data bus. Moreover, gateways (bus bridges) can easily be designed to be sufficiently fast to reduce latency, thus the time delays would only be related to waiting for the global data bus access.

#### 2.2.1.3. Applicability for reconfiguration

In order to perform efficient reconfiguration one would require a number of non-dedicated modules capable of running any downloaded program. In architecture "A", however, all modules are dedicated to performing certain functions and are closely bound to them. Although a big number of processing modules per cabinet gives great potential for reconfiguration, the modules specialisation violates constraints of reconfiguration and diminishes the applicability of the design "A" for dynamic reconfiguration.

In addition, the implied use of many dedicated I/O modules in a cabinet restricts the applicability of the global reconfiguration approach. For example, a function performed in cabinet X requires a dedicated I/O module that is unavailable in cabinet Y, so this function could not be executed in cabinet Y, even if the required program was successfully downloaded. This problem could be solved, or at least reduced, by the use of remote data concentrators (RDC) employed to pass signals from devices using the global data bus as the communication media.

If the strict dedication of processing units to avionics functions is abandoned, the use of similar hardware for redundancy purposes gives a possibility for some reconfiguration (the same design of modules makes them suitable for interchanging software).

In view of the above discussion the design of architecture "A" has to be branded as not especially attractive for implementation of RIMA.

## 2.2.2. ARINC 651 architecture "B"

### 2.2.2.1. Overview

Architecture "B" proposes very powerful and fault-tolerant processing modules as a core for every cabinet. Each cabinet includes only one processing element with dedicated I/O, data bus interface and a power supply (see Figure 2.2 below). The number of cabinets is not defined and will vary depending on system requirements.

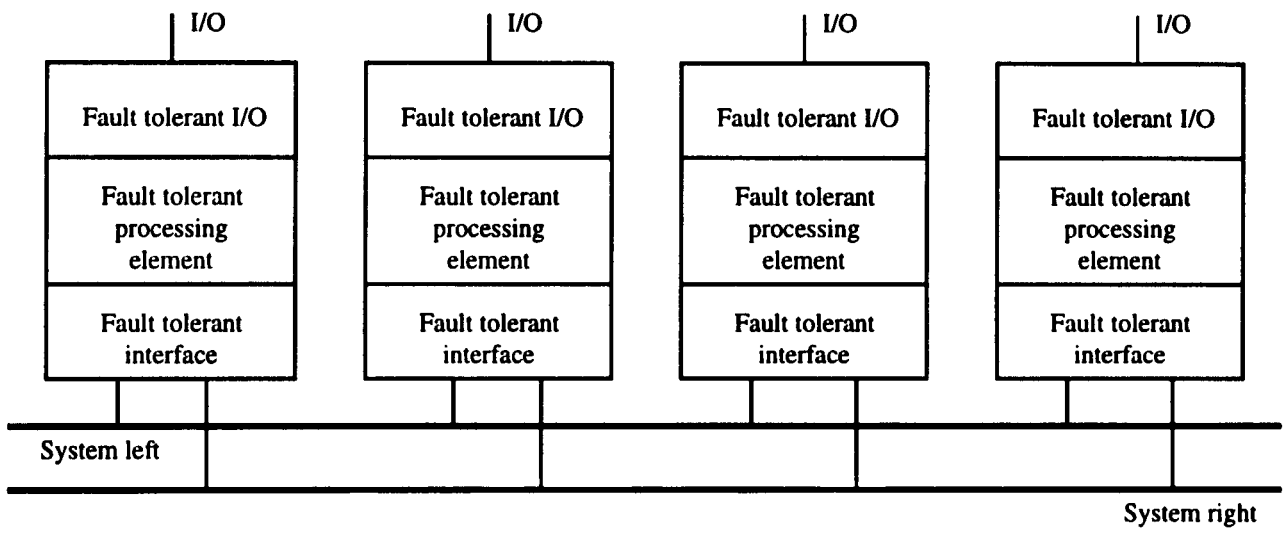



**Figure 2.2. ARINC 651 IMA architecture example "B".**

All the cabinet processing is performed by only one core module executing multiple applications. Since there are different programs running on each processing unit, memory management is implemented in hardware to provide good software partitioning and transparency for applications. The software executive is responsible for the scheduling and control of executed programs. The core module is fault-tolerant; this is achieved internally by redundancy of processor components (minimally a pair of dual redundant processors). Analogously, the design of I/O and power supply modules aims for fault tolerance; that is implemented by similar hardware redundancy, so that a cabinet as a whole can be considered fault-tolerant. On the system level, redundant cabinets provide additional fault tolerance and system availability.

#### 2.2.2.2. Advantages and shortcomings

At this place several features of the architecture being discussed are presented with emphasis on their beneficial characteristics or associated drawbacks.

Advantages:

- The architecture "B" design aims to achieve a great level of fault tolerance across the whole system on every level (module, cabinet, system). Due to internal redundancy (which is transparent for applications), the expected mean time between failures (MTBF) for the core LRMs can be very high and thus requiring maintenance in long time intervals.
- The demand for spares with this approach is very minimal, since there are few types of modules (core, power supply and a few standard and specialised I/O modules).
- Hardware implemented memory management makes the memory of different applications impenetrable for one another and thus eliminates a possible danger of software errors affecting different avionics functions. Different avionics sub-systems are independent even if their functions are executed by a single module.

Shortcomings:

- Although the processing modules are standardised (and they could be moved between cabinets), since there is no means for core LRM interchangeability within a cabinet, the main constraints of the IMA approach seem to be violated. With this architecture a cabinet can be thought of as an equivalent of a black box (LRU) in the traditional approach to avionics systems design. The only standard modules that can be maintained separately and interchanged within a cabinet are the I/O and power supply modules. However, the close physical dependence between I/O modules and core modules renders even this task rather difficult.
- Highly sophisticated and complex core LRMs are likely to be very expensive. The market for such complicated units can be expected to be rather limited, and the number of manufacturers willing to design and produce the LRMs also to be very small.[5]

[5] The flight control computer on a Boeing 777 can be thought of as an example of such a highly complex fault-tolerant module, and its price is estimated to be some US $600,000. Due to the sensitive nature of this information the price can only be estimated based on informal discussions with various aerospace related companies.

- Although fault-tolerant, in case of a major failure of the core module, the system will encounter a significant degradation of operation due to the loss of many different functions. That could possibly affect many avionics sub-systems.
- Authors of the design claim the system determinism to be one of its features, which could simplify system certification. However, when concurrent scheduled computing systems are considered, formal methods for proof and verification are likely to be unfeasible in practice. This will be even more complex as the set of tasks being executed by a particular unit changes due to reconfiguration.

#### 2.2.2.3. Applicability for reconfiguration

In principle, reconfiguration takes place when a core module undertakes a function previously performed by another module. In the case of an avionics system based on design "B" there is no immediate possibility of a single function loss. A failure of a core LRM would be followed by a loss of many functions that would have to be undertaken by other modules. Since, however, there is only one core module per cabinet, the reconfiguration could take place only between cabinets. That would imply that every cabinet would have to be able to store programs for all avionics functions, and thus the non-volatile memory requirements would grow significantly. Alternatively some additional software storing modules would have to be added to the system.

The method for reconfiguring many functions at the same time (equivalent to simultaneous failures of many core LRMs in architecture "A") is likely to be complex, and so, difficult to be proven correct or tested. Also the traffic on the ARINC 629 global data bus will be significantly increased due to messages related to reconfiguration.

Theoretically there exists a possibility of a single function loss due to a fault in the memory management hardware. In that case, the affected core module could reconfigure itself to preserve the most critical functions. However, since the memory management hardware is built into the processing unit, its fault would probably affect all the tasks being executed leading again to the loss of multiple applications. On the other hand, if the memory management had not been integrated with the processor, a fault of the memory management software or hardware would still be of a critical nature and could lead to some undesirable interference between processes and to highly critical modes of failure.

## 2.2.3. ARINC 651 architecture "C"

### 2.2.3.1. Overview

In architecture "C" based systems, processing is distributed between several cabinets. Each cabinet employs multiple core modules to provide the processing power, a number of gateway/bus bridge modules for interfacing the global ARINC 629 data bus, power supply modules and a varying number (possibly none) of dedicated I/O modules. All modules are connected to the ARINC 659 backplane bus and all inter-module communications are handled on the backplane bus (see Figure 2.3).

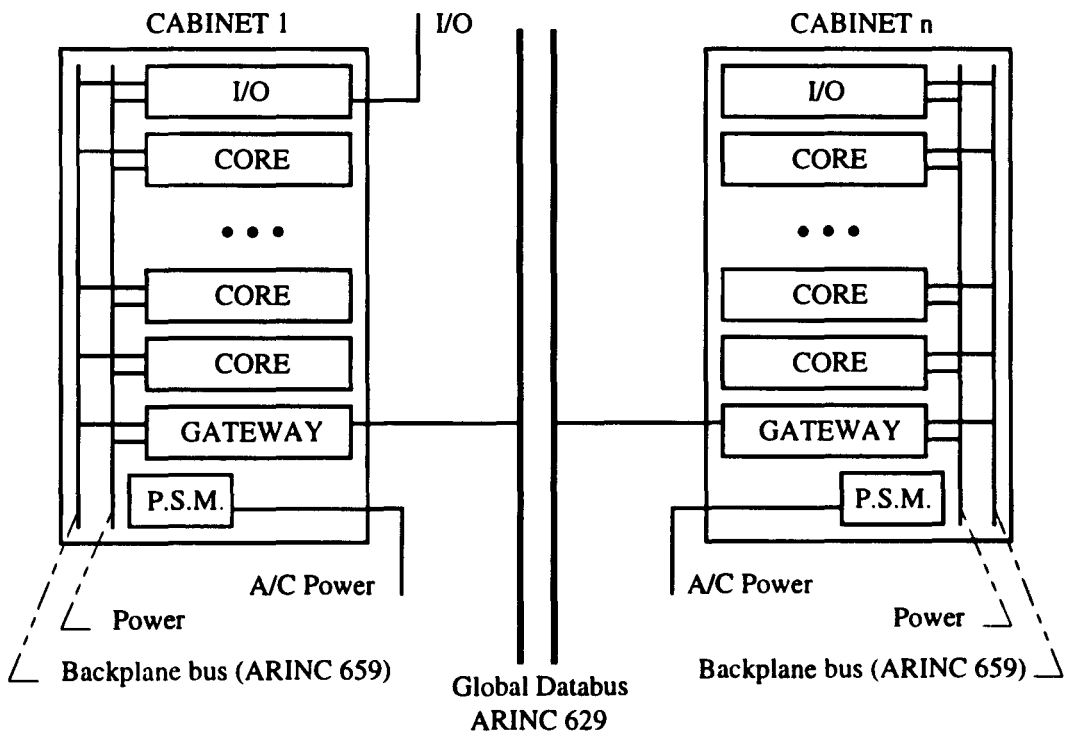



**Figure 2.3. ARINC 651 IMA architecture example "C".**

All the processing in core modules is based on single executive - multiple application (SEMA) basis. Several avionics functions can be executed by a single module. However, one difference to the architecture "B" core module is that it does not have to be as powerful, and multiple core LRMs are employed to provide the functionality of a cabinet. Also unlike the previously discussed design, architecture "C" processing elements are physically separated from their I/O interface.

Similarly to the previous example, the memory management in core modules is supported by dedicated hardware built into the processor. This technique makes the memory partitioning robust and transparent for software applications.

Although it is not shown in Figure 2.3 (above), the cabinet components such as power supply, I/O and gateway modules are duplicated for redundancy to achieve the necessary level of fault tolerance. The ARINC 659 backplane bus and ARINC 629 global data bus are duplicated in an analogous manner.

#### 2.2.3.2. Advantages and shortcomings

In this section various features of the architecture being discussed are presented with emphasis on their beneficial characteristics or associated drawbacks.

Advantages:

- The architecture design points to the employment of RDCs and smart actuators as a possible way to avoid the unnecessary use of dedicated I/O modules and to reduce the complexity of system upgrades. Cabinets without I/O modules acquire their data from the ARINC 629 data bus, and thus their location on the aircraft can be freely chosen. On the other hand, a dedicated I/O module should be placed relatively close to its data source/sink in order to reduce discrete wiring. This could be perceived as an undesirable limitation.
- The use of a physically independent I/O module for acquiring analogue data, converting it to a digital form and making it available on the backplane bus, makes the core modules fully interchangeable and independent from their avionics functions. There is no dedication of core modules to any particular avionics functions.
- Hardware implemented memory management makes the memory of different applications impenetrable for one another, and thus eliminates the danger of software errors affecting different avionics functions.

Shortcomings:

- A generic problem of possible bottlenecks that could be encountered when gateway modules interface different speed data buses is present in this design. This problem was also present in architecture example "A".
- Some new common mode failures have to be considered in relation to the use of gateway modules. Since gateways are responsible for passing data to and from many core LRMs that may perform

functions from different avionics sub-systems, their failures could affect many avionics functions and lead to unacceptable system degradation. Therefore, the gateway modules will have to be fault-tolerant by design and should be duplicated for redundancy.

- The possibility of executing multiple applications by any core module introduces the danger of a loss of many functions due to a failure of a single module. Also inter-dependence between different avionics sub-systems may occur if one core LRM is to perform functions belonging to distinct avionics.
- Due to its high complexity and integration, the certification of a system based on multiple application processing modules can be difficult, and the formal methods are unlikely to be employed.

#### 2.2.3.3. Applicability for reconfiguration

The use of multiple undedicated core modules per cabinet gives a certain potential for the local reconfiguration approach. The modules could undertake each other functions in case failures, that could be stored internally in the module non-volatile memory. A detailed analysis should be performed to establish the amount of non-volatile memory required to store all functions performed within the cabinet.

The multi-application approach to processing may introduce some problems to the reconfiguration domain. A failure of a single module will cause the loss of several functions, so the reconfiguration method gets more complex, and its proof of reliability becomes more difficult. That could be resolved by abolishing multitasking core LRMs in favour of a greater number of simpler modules.

Unlike in the case of architecture "B", the messages related to reconfiguration will travel on the backplane bus, which provides much greater capacity than the ARINC 629 bus. Thus, the reconfiguration algorithm will not introduce a significant relative growth of traffic on the bus as in example "B".

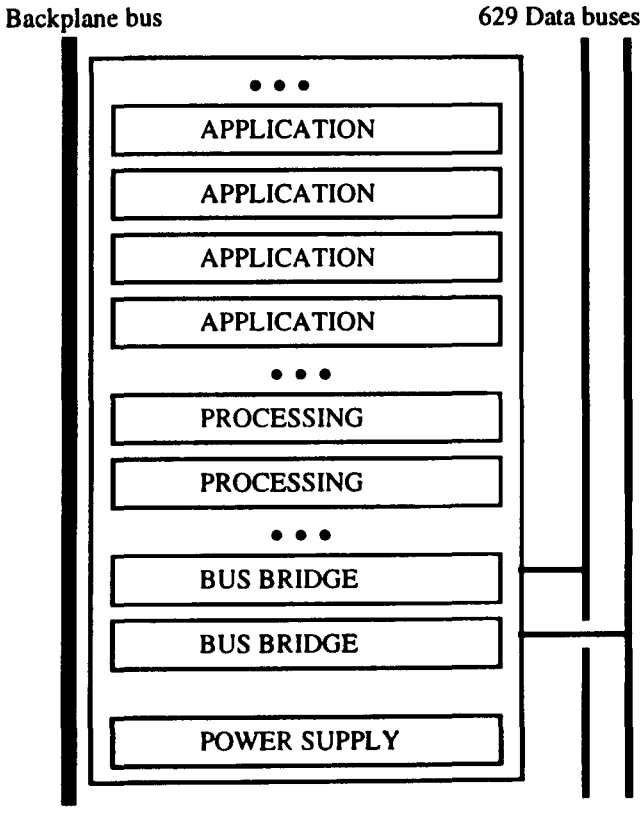
Architecture "C" gives quite good opportunities for reconfiguration with respect to both local and global approaches. In order to reduce the complexity of the reconfiguration algorithm, some changes to the processing model implemented by the core LRMs could be required. This would most likely involve a replacement of the single executive multiple applications model with the alternative single executive single application implementation. If the amount of non-volatile memory necessary to store all required

functions in a module is too big, an external software store device could be added and some means of retrieving the programs provided.

## 2.2.4. ARINC 651 architecture "D"

### 2.2.4.1. Overview

Architecture "D" achieves its processing power by employing a number of processing modules. Each module is capable of downloading and executing any application that can be executed concurrently, so that one module can support multiple applications. The software for the avionics functions is stored in separate application modules using their non-volatile memory. There is no notion of dedicated I/O modules and all data transfer is based on the ARINC 629 data bus with the use of bus bridge modules for data exchange between global and backplane data buses (see Figure 2.4 below).




The number of core processing modules is not defined by the design and depends on the required processing capacity. Each core LRM includes interfaces to the power supply and the ARINC 659 backplane bus, application controller, some small non-volatile memory for storing software executive and large RAM for executing application programs. Memory management is hardware implemented (similarly to examples "B" and "C"), and provides robust and impenetrable software partitioning. The software executive is responsible for scheduling, controlling and executing multiple applications.

**Figure 2.4. ARINC 651 IMA architecture example "D".**

The application modules include large non-volatile memory for storing application programs and data, a central processor and memory management unit, small RAM for handling application status information,

ARINC 659 I/O buffer and interfaces to power supply and ARINC 659 backplane bus. Each application module can store programs for one or more avionics functions.

On the cabinet level fault tolerance is provided by replication of application programs on redundant core LRMs, that if necessary, can be based on dissimilar hardware. On the LRM level fault tolerance can be achieved by internal fault detection and isolation, and by providing redundancy of tasks performed by the core. Redundant copies may be of different design and implementation to avoid generic errors.

To ensure an even greater level of fault tolerance, architecture "D" provides the capability for dynamic reconfiguration by means of a reconfiguration function. In the case of an application module or processing module failure, the reconfiguration function - with the use of reconfiguration strategy tables - tries to allocate affected applications to available processing modules either internally in the cabinet, or externally to other cabinets. Moreover, based on a flight phase it could determine which applications are unnecessary and withdraw them from execution (e.g. braking function during cruise).

#### 2.2.4.2. Advantages and shortcomings

In this section various features of the architecture being discussed are presented with emphasis on their beneficial characteristics or associated drawbacks.

Advantages:

- Since each core module provides a general capacity for processing and is capable of supporting any necessary function, there is no notion of dedication between an LRM and an avionics function. Moreover, tasks executed by a core LRM are attributed to the module at system start-up rather than during the design phase.
- The set of avionics functions available within a cabinet depends only on application modules. This adds more flexibility to the system and cabinets that - being capable of processing any required software - can be configured for re-use even on different aircraft. This, when combined with the lack of any dedicated I/O modules, renders the cabinet fully functionally independent.

- Cabinets free of any dedicated I/O modules can be placed in any maintenance accessible place and thus leave more space for pay-load volume. Also, one could expect a reduction in discrete wiring following the lack of any I/O modules inside the cabinets.
- As in the case of IMA architecture example "B", hardware implemented memory management makes the memory of different applications impenetrable for one another, and allows the elimination of some of the possible problems with memory consistency.
- The possibility of dynamic reconfiguration is embedded in the architecture design, and makes it rather attractive for implementation as reconfigurable IMA.

Shortcomings:

- The possibility of executing multiple applications by any core module introduces the danger of a loss of many functions due to a failure of a single module. Also inter-dependence between different avionics sub-systems may occur if one core LRM is to perform functions interacting with distinct avionics.
- Due to the high integration and complexity of systems based on multiple applications processing modules, the certification of such systems can prove to be difficult, and the formal methods are unlikely to be employed.
- The generic problem of interfacing different speed data buses can again be observed in this design, as it employs bus bridge modules for data exchange between the backplane and system data buses.
- The bus bridge modules – which are equivalent to gateway modules in the previous design (example "C") - should be considered equally critical, and similar observations about their fault tolerance are valid in this case.

#### 2.2.4.3. Applicability for reconfiguration

Although a possibility for dynamic reconfiguration is provided as a part of the architecture design, some problems may arise.

Firstly, design authors seem to dedicate the ARINC 659 backplane bus to download application software to processing elements. However, since the backplane bus will also be used for inter-module communications and for signal handling to and from the cabinet, its capacity is unlikely to be sufficient.

Moreover, as any single core LRM could execute multiple avionics functions, its failure would lead to the need for the reconfiguration of many applications at the same time, and thus it would introduce a serious traffic problem on the ARINC 659 backplane bus. An additional software downloading bus could possibly solve this problem. Secondly, the method for reconfiguring many functions at the same time (following a failure of a multiple application core LRM) is likely to be complex and its formal proof difficult or unfeasible.

Despite these shortcomings, one has to emphasise some of the architecture features that make it very attractive for reconfiguration. All necessary software is already available in the cabinet, possibly with some redundancy. Also, since application modules are capable of storing the application state, reconfiguration of functions requiring state information might prove to be somewhat easier.

Architecture "D" looks very promising for as a basis for a reconfigurable IMA system.

### 2.2.5. ARINC 651 architecture "E"

#### 2.2.5.1. Overview

The cabinets for architecture "E" include core processing modules, standard and specialised I/O modules, bus bridge or gateway modules and power supply modules as in Figure 2.5 (below). Modules within a cabinet communicate via the ARINC 659 backplane bus, and the communication between cabinets is based on the ARINC 629 global data bus. The number of core LRMs and dedicated I/O modules in a cabinet can vary depending on the actual system requirements.

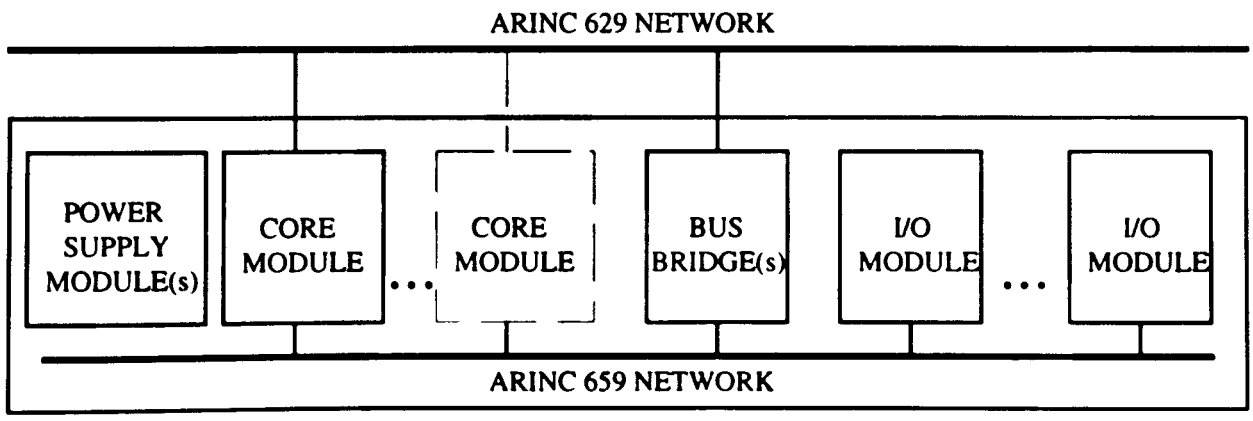



**Figure 2.5. ARINC 651 IMA architecture example "E".**

Core modules provide processing capacity for multiple applications, and it is the executive software that enforces strict segregation between applications. They also maintain application scheduling, health monitoring and inter-application communications. Programs executed by a module and their data are stored in the module non-volatile memory, and RAM is used for actual program execution. Core module hardware is standardised and independent from the executed program. The avionics functions performed by a processing module depend only on the software downloaded during cabinet configuration. There is no provision for dynamic reallocation of resources and the distribution of application is decided during system design and integration.

Applications performed by a single module share some global memory to facilitate their mutual communications. Also each application has its own memory space guarded from other applications by the executive software.

#### 2.2.5.2. Advantages and shortcomings

At this place several features of the architecture being discussed are presented with emphasis on their beneficial characteristics or associated drawbacks.

Advantages:

- The use of independent I/O modules for acquiring analogue data, converting it to a digital form and making it available on the backplane bus, makes the core modules fully interchangeable and independent from their avionics functions, however it could constrain the set of functions performed within the cabinet. There is no presumed dedication between core modules and avionics functions.
- The possibility of a direct connection of a core module to the global data bus (ARINC 629) reduces the problem of a possible bottleneck on gateway/bus bridge modules. Also, bus bridge modules may become less critical as they are no longer responsible for passing all the data to and from the cabinet. However, a new problem may arise when a core LRM processing has to be suspended due to waiting for an access to the ARINC 629 data bus. That could be avoided if data bus transactions were handled independently of the main processing element.

Shortcomings:

- The possibility of executing multiple applications by any core module introduces the danger of a loss of many functions due to a failure of a single module.
- The executive software based application partitioning does not seem to be as strong as in the case of the hardware equivalents (architectures "B", "C" and "D"). Executive software has to be developed most carefully in order to eliminate any generic errors. A failure in memory management may lead to a loss of many avionics functions possibly affecting many avionics sub-systems.
- Due to various memory consistency issues, the certification of a system based on multiple applications processing modules may become even more difficult in the case of a shared memory system. Applications communicating with the use of the global memory interface introduce the danger of errors affecting many avionics functions or sub-systems.
- The large number of different types of I/O modules introduces unnecessary specialisation to the cabinet. Also, the placement of cabinets is no longer unlimited due to consideration of discrete wiring necessary for signal handling from data source to an I/O module and back to its data sinks.

#### 2.2.5.3. Applicability for reconfiguration

The design does not provide any means for resource reallocation, and the situation gets even worse when the use of specialised I/O modules has to be considered, as the specialisation significantly reduces the applicability of global reconfiguration approaches.

The certification of a reconfigurable system based on architecture "E" might also prove difficult due to the major role of the software executive, and the multiple applications performed by any module. A failure of a core module would involve the need for reconfiguring many functions at the same time (equivalent to simultaneous failures of many core LRMs in architecture "A"), which is likely to be complex, and thus difficult to be proven or tested.

## 2.2.6. Discussion

### 2.2.6.1. Common features and generic problems

Although the five examples of IMA architecture discussed in this chapter differ, one can identify certain common features, since they all implement the idea of IMA. Also, there are several generic problems that, as shown in the ARINC 651 IMA architecture examples, can be solved in many different ways.

- **Core modules and computation models.** The designer can base the processing model on a Single Executive Single Application (SESA) approach or provide an operating system for parallel or concurrent computation. The processing capacity of a core LRM can vary from a simple, single application computer (ARINC 651 example "A") to highly complex, multiple application, fault-tolerant modules as in example "B". Also, the choice has to be made as to whether the modules are to be based on similar or dissimilar hardware in order to achieve the required fault tolerance.
- **Buses.** It seems inevitable that ARINC 629 will be employed as the global system data bus. Although the backplane bus is most likely to be implemented according to the ARINC 659 specification [9], there exists a possibility for employing some alternative solutions. For example, current work on IMA carried out by British Aerospace use FDDI as a backplane bus for demonstration purposes (see Chapter 9). Some problems may arise due to the synchronous nature of the ARINC 659 data bus. The ARINC 653 report strongly suggests that all core modules should be synchronised by the backplane bus; that may be unfeasible as synchronisation of many, possibly dissimilar, processing elements can be complex and difficult. On the other hand strict synchronisation of communications could enforce the deterministic behaviour of the cabinet as a whole, that could simplify testing and certification of such systems.
- **Interfacing the global data bus.** The problem of data exchange between cabinets and remote devices is common to all designs. Some solutions propose bus bridge/gateway modules to provide an interface between the backplane and the global data bus, while some provide a possibility of direct core LRM - global data bus communications.
- **Dedicated I/O modules.** Most of the IMA architecture examples proposed in ARINC 651 employ some form of I/O modules. They can be either core LRM dependent (as in example "B") or can be implemented as totally independent modules to convert analogue signals to their digital form and transfer them onto a backplane bus. The architecture example "D" avoids possible problems related to

I/O modules with the use of smart actuators and remote data concentrators that provide all the required signals in the digital form on the ARINC 629 global data bus.

#### 2.2.6.2. Prospects of dynamic reconfiguration

From the discussion on the ARINC 651 IMA architecture examples, several features that make reconfiguration unattractive or unfeasible can be established.

- Having a small number of cabinets in the system decreases its flexibility. The possibility of a "free" assignment of different functions to different cabinets would be significantly reduced if the system functionality was to be provided by just a few cabinets. That would have a serious impact on the applicability of any global reconfiguration method. On the other hand, however, an increased number of core modules per cabinet increases the applicability of local reconfiguration schemes, which seem to be more attractive than any global approach (see sections 1.2.1 and 1.2.2).
- An option employing very powerful processing modules to provide the functionality for the whole cabinet, immediately disables any local reconfiguration scheme. Also, in case of a loss of such a core module, there would be a need for reconfiguration of many of the cabinet functions (excluding only functions whose loss would lead to some minor or possibly major failure conditions). That would significantly increase the complexity of the reconfiguration scheme, and the traffic on the global data bus (the backplane buses have to be excluded when the only applicable approach is global reconfiguration). This IMA architecture, however, may be very attractive if reconfigurability is not required (IMA rather than RIMA).
- In the case of core LRMs executing multiple applications (SEMA), the system has to face a danger of a loss of many functions due to a failure of a single module. To reduce this danger, core modules would have to be internally fault-tolerant (that would increase their complexity and price), or some dedicated redundancy would have to be employed. Generally with this core LRM design reconfiguration methods become complex and difficult to prove correct. Thus, the SEMA model of computation seems to be less practical for reconfiguration than the single executive single application model (SESA).
- Any dedication of a core LRM to an avionics function reduces the applicability of any reconfiguration approach. When there are no means for in-flight software downloading (where a function is

permanently[6] assigned to a core module, as in example "A"), the dynamic reconfiguration is simply impossible. However, static on-ground reconfiguration could still allow the aircraft to be dispatched even if some modules are off-line.

> Commentary
> This of course raises questions related to FAA/CAA regulations forbidding dispatch of an aircraft with a known failure, and is likely to be considered as one of a number of certification issues regarding reconfigurable IMA systems.

- The use of a backplane bus for downloading software is not desirable for the purposes of dynamic in-flight reconfiguration. The ARINC 659 data bus is generally not fast enough to provide the necessary capacity for supporting inter-module communications, data transfer to and from modules and software downloading. Moreover, employing a highly critical resource – the backplane bus – for downloading software would introduce some additional safety hazard to the system as a whole.
- Dedicated I/O modules introduce unnecessary specialisation to the cabinet. The choice of functions performed within a cabinet is restricted by the I/O requirements and the availability of I/O modules. The global reconfiguration is unfeasible, and only the local approach could be implemented. Also, dedicated I/O modules constitute possible points of failure, and as such would have to be designed to be fault-tolerant and appropriate redundancy would have to be employed.

Several of the features of the IMA architecture examples discussed in this chapter can be described as highly desirable for reconfiguration purposes.

- The SESA model of computation - proposed in example "A" - implies a loss of a single function in case of a core LRM failure. This is a preferable model when dynamic reconfiguration is to be considered. Reconfiguration schemes based on reconfiguring a single function in case of a single failure are likely to be much less complex than any method applied to a SEMA model, and as such more suitable for formal proof methods. Note that the term "function" could also describe a group of functions being inseparable (always reconfigured together). Such defined "functional groups" could be then treated as single applications, and the reconfiguration method would see the SESA model instead of SEMA.

---

[6]Function assignment could be changed only as a result of some static reconfiguration procedure, that would not be applicable during an aircraft flight phase.

- Standardised general purpose core modules based on similar hardware constitute a good feature for reconfiguration purposes. The lack of any dedicated core LRMs gives a great potential for local dynamic reconfiguration schemes. This can be further extended to global methods if there are no dedicated I/O modules in the cabinet, and thus a set of functions performed within the cabinet can be relatively freely chosen.
- Application modules, as proposed in architecture "D", increase the applicability of dynamic reconfiguration to an IMA architecture. In case of a failure of a core module, its function can be undertaken by another core LRM, and the necessary software downloaded from an appropriate application module. Possible replication of software across many application modules gives a necessary level of fault tolerance. Moreover, since application modules are capable of storing function state, the task of state preservation may prove somewhat easier to accomplish. Also, an application module could store a local database necessary for some functions (e.g. navigation).
- An extensive use of RDCs and smart actuators leads to a reduction or even elimination of the need for any dedicated I/O modules (example "D"). As mentioned above any I/O module present in a cabinet introduces unnecessary specialisation and restricts functions that can be performed within this cabinet.
- The possibility of a direct connection between a core module and the ARINC 629 data bus can increase the applicability of a global reconfiguration method. In the case of global reconfiguration, some extra traffic can be expected on gateways or bus bridges that can be easily avoided if core modules are directly connected to the system data bus. However, since even the ARINC 659 data bus seems to be too slow for dynamic in-flight software downloading, it is highly unlikely the ARINC 629 bus – which is about fifteen times slower - would be employed for that purpose.

#### 2.2.6.3. Reconfigurable IMA

Having established the desired features of a Reconfigurable IMA architecture one could try to adopt the most promising architecture examples for the purpose of reconfiguration. In this section the design changes necessary to adapt examples "C" and "D" to reconfiguration are discussed.

***Core modules***

As discussed above, the SESA computational model is preferable to the SEMA approach when dynamic reconfiguration is to be considered. That would also significantly reduce the need for processing capacity

in a single module. However, when processing requirements of applications differ significantly, some core modules would not fully exploit their processing power.

In SEMA based core modules, some problems may occur with deterministic scheduling and memory management when the set of applications performed changes due to reconfiguration. Each application has its own processing requirements, that define its time slot in the overall processor time. Similarly, memory requirements restrict the set of functions that can be performed simultaneously on a single core LRM. In this situation "free" reconfiguration of one application to another may change the overall processor power and memory requirements, and thus it may require a change of the scheduling algorithm or it may even render execution of the set of functions unfeasible.

The SEMA model of execution could be employed in RIMA, provided that whole function subsets[7] rather than single functions are to be reconfigured. That, however, would imply grouping functions with regard to their criticality which may not be desirable in some systems. For instance, if critical functions have very high processing requirements no more than one of them could be performed by a single core LRM. Alternatively, a failure of a module performing a set of critical applications could prove to be too severe to handle by aircraft systems even for a very short time required for reconfiguration.

*Software store*

The application software has to be available at any time for any core module to make dynamic reconfiguration possible. In architecture "D" the software store is provided explicitly in the form of application modules. Each module can store one or more applications to provide a necessary level of redundancy (software replication across a cabinet). It is also easy to realise that the main difference between examples "C" and "D" is the presence of application modules in the latter one. Introducing application modules to example "C" would simply convert it to some variation of example "D". To keep those two distinct, some other means for software storing has to be designed for architecture "C".

[7] Hence, application as a generic term could mean either a single function or a group of functions of similar criticality that are reconfigured as a whole.

A relatively simple solution could implement the software store directly on the core LRM with the use of some programmable non-volatile memory. Simple analysis shows that the reconfiguration would be fastest if every module was capable of storing locally the software for all avionics functions performed in the cabinet. However, if this approach is unfeasible (either prohibited for economical reasons or physically impossible), each module could store software just for a few applications, and thus provide a similar level of application replication as in the example "D". In this design, two distinct modules from architecture "D" (core and application) are joined to provide the processing power and software store as a single physical unit.

***Software downloading***

The module capacity for software downloading is required by two distinct phases of system operation. Firstly, at the system start-up, core modules have to download the software for functions they are to perform. This phase tends not to be time critical. Secondly, if some core LRM has to change its application from A to B due to reconfiguration, the program for application B has to be downloaded in a time critical manner. A design where core modules store all the necessary applications software internally could, therefore, prove to be the best option.

Clearly, in-flight software downloading is not necessary if - as suggested in one of the variants of reconfigurable architecture "C" - every module has the capacity for storing all applications. However, if this is not the case, some other media than a backplane bus would have to be provided. The ARINC 653 report [10] argues strongly that the backplane bus has to be used for downloading software. However, as already emphasised, it is unlikely that the ARINC 659 data bus will have a high enough capacity to provide these services, thus more appropriate alternatives might be required.

***Signal handling***

Any dedicated I/O modules introduce unnecessary limitations to the cabinet design. An optimal RIMA architecture should focus on the possibility of handling all signals with the aid of the global data bus. Remote data concentrators and smart actuators could be of a great assistance in completing this task.

In systems without dedicated I/O there is no need for specialised LRMs, and thus the effort on module design could be reduced, although some additional work would then be required to develop appropriate RDCs and smart actuators. A cabinet design based on the elimination of any dedicated I/O should be simpler, more generic and suitable for different types of aircraft.

#### 2.2.6.4. Conclusion

Based on the discussion presented in this chapter IMA architecture examples "C" and "D" have been identified as the most promising for implementation of a Reconfigurable IMA system. It is clear that in both cases, some alterations will be required to the ARINC 651 proposed design, however, efforts have to be made to preserve the original outline in order to minimise the costs of initial system development.

## 2.3. RIMA - proposed architecture

In this section two distinct RIMA architecture examples are described and discussed. These are adaptations of the two most promising IMA approaches described in the ARINC 651 specification ([2] section 6, examples "C" and "D"). In both cases the system design is qualitatively analysed with respect to possible points of failure and related failure modes.

### 2.3.1. Architecture based on ARINC 651 example "C"

#### 2.3.1.1. Design overview

The architecture shown in Figure 2.6 (below) follows the specification of ARINC 651 example "C". Although there are many similar features some aspects of the design have had to be changed in order to make it more suitable for dynamic reconfiguration.

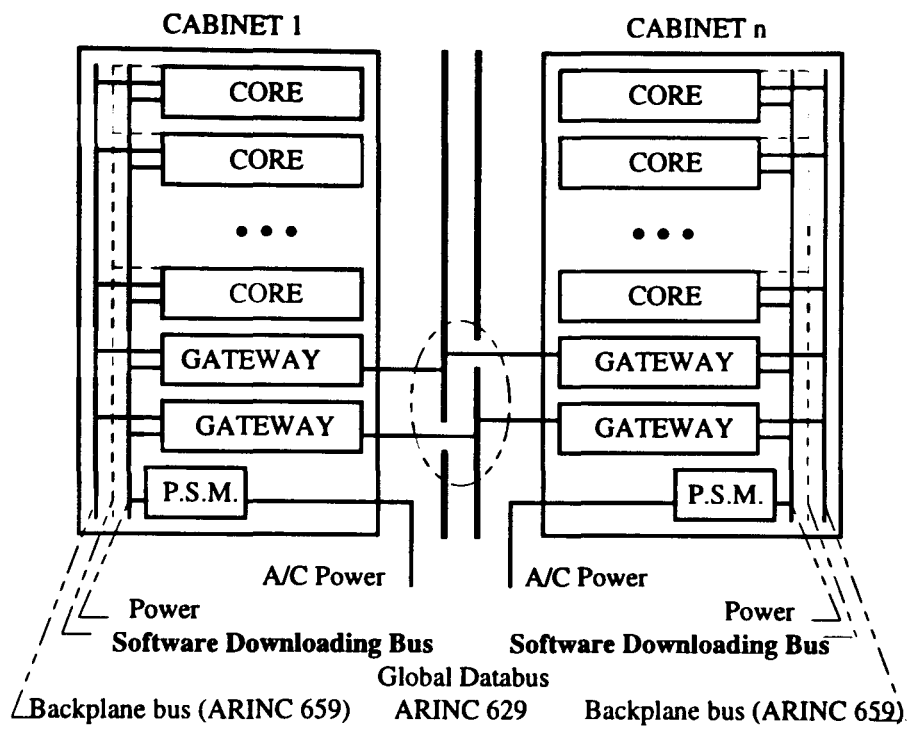



**Figure 2.6. Reconfigurable IMA, based on ARINC 651 architecture "C".**

### *Modules*

Core modules, apart from providing the processing power, also provide means for software storing and application state updating. To complete this task all core modules consist of: a processor, large non-volatile memory[8] to store applications software and function states, RAM for program execution and interfaces to data buses and power supply.

The core module processing model is based on single executive single application bases. In such an approach a single module failure leads to reconfiguration of at most one function at a time[9].

> Commentary.
> The single executive multiple applications approach could be adopted for this design if certain restrictions on applications performed by a single module were made. Each function belonging to a set performed by a single module would have to be of identical or similar criticality. That constraint would allow the reconfiguration of the whole functional set in the case of a core LRM failure, so that the

[8] Depending on the reconfiguration algorithm and required level of software replication the non-volatile memory requirements may vary. In the most exhaustive case the module would have to store the software of all functions within the cabinet. In the case of less robust approaches there may be a need to store just a few functions on each module.

[9] If the level of function replication is not great enough there may occur a case of some form of chain reconfiguration, i.e. module A takes the function performed so far by module X, while the former function of module A has to be undertaken by module B.

reconfiguration algorithm would still see the SESA core module and treat a group of applications as a single one.

The state related to some avionics function is updated by every module storing this function. The update mechanism can be activated by an appropriate message on the backplane bus.

Commentary.
To preserve function state even in case of temporary lack of power some non-volatile memory should be used for that purpose. Typical non-volatile memory has limitations on the number of write cycles, and thus it is not suitable for frequent updates, thus CMOS RAM with battery back-up or similar non-volatile memory could be used instead. These types of non-volatile storage are not suitable for storing all the application software due to their high price and capacity limitations. However, since function state information is expected to be relatively small, this solution could prove to be feasible.

All core modules should preferably be implemented on the same hardware/software platform (having the same processor and operating environment would allow application software to be executed by any core LRM), although their internal design and manufacturers may differ (that would prohibit generic faults due to errors in the module design).

Gateway modules perform the same functions as described in ARINC 651 ([2] section 6.6.3). The use of gateways leads to better encapsulation of the cabinets functions and their independence of any external environment (core LRMs communicate only with the use of the backplane bus).

Some attention has to be paid to the way of connecting gateways to global data buses. IMA architecture example "C" suggests a connection of gateway modules to two ARINC 629 data buses as shown in Figure 2.6. With such a connection, in case of a failure of one of the ARINC 629 data buses and one of the gateways, there could occur a situation in which a cabinet is physically disconnected from the system. If this solution does not provide the required level of fault tolerance (subject to quantitative analysis), a connection as in Figure 2.7 (below), could be used instead. In this case either both gateways or both data buses would have to fail for a cabinet to be lost.

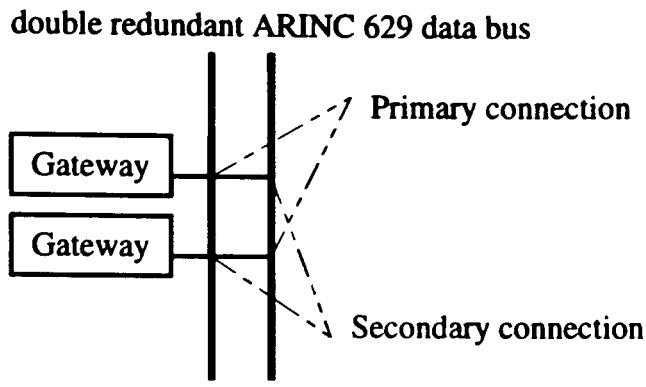



**Figure 2.7. Fault-tolerant gateway - ARINC 629 data bus connection.**

Connecting each gateway to both buses brings an immediate benefit in fault tolerance. In the case of a gateway module failure, the other gateway can transmit signals to both buses, so that a subsequent failure of a single bus would not affect the system performance. Note that a secondary connection becomes active only if the corresponding primary connection of the other gateway becomes inactive, and thus, such a solution should stop a faulty gateway module from bringing down both buses (e.g. by continuously trying to transmit).

As discussed in sections 2.2.6.2 and 2.2.6.3, any dedicated I/O is not desirable when reconfiguration is being considered. Thus, unlike in the ARINC 651 example "C", there are no dedicated I/O modules in this design. All signals are handled on the ARINC 629 and ARINC 659 data buses.

***Buses***

The global system data bus and the backplane bus are implemented as proposed in ARINC 651 ([2] section 6.6), although the actual specification of the backplane bus could change subject to available alternatives.

The software downloading bus should be implemented as a high throughput, possibly optical fibre based data bus. Its protocol should be optimised for the burst mode transfer with the purpose of shortening the downloading time.

> Commentary
> There would be no need for a software downloading bus if every module was capable of storing all the necessary software. Moreover, appropriate (not necessarily full) replication of software across the cabinet could eliminate the need for any software downloading bus. Provided that the number of replicas meets the

FAA/CAA safety requirements [3], the software downloading bus could be considered optional (hence the dotted line in the design overview figure - Figure 2.6).

#### 2.3.1.2. Points of failure

In this section possible safety hazards related to failures of different system components are discussed. The analysis is qualitative and focused on the cabinet level.

*Core modules*

A failure of a single core LRM leads to a loss of an application (one or more avionics functions). Depending on the criticality of the lost function the reconfiguration mechanism can either start actions to allocate the function to some other core module, or alternatively can accept it as system degradation. In the case of the need for reconfiguration, the system will suffer a loss of some other function (the least critical one in the cabinet), and thus some system degradation will follow any single core LRM failure. Also, as core modules store applications internally, a decrease of the level of software replication and thus the level of system reconfigurability follows each failure of a core LRM.

A loss of a function could be postponed if some dedicated redundancy was provided, and thus there would be no system degradation following the first few core LRM failures (the actual number would clearly depend on the number of spare core modules). Such an approach, although not fully exploiting system reconfigurability, would provide greater fault tolerance and would be more attractive from the certification point of view since the system would provide 'dedicated' redundancy for its most critical functions.

If the SEMA approach is chosen, several avionics system can be affected by a single failure. In this case the failure mode is a function of the criticalities of the lost avionics applications, and is likely to be more severe than the criticality of any single one of them. To assess the failure modes related to any core LRM failure, the actual groups of functions executed by a single processing module would have to be known. This, however, depends on the aircraft type and is not generic to RIMA.

Since core LRMs are also used to store functions, in the case where memory limitations do not allow full replication of the required software, the reconfiguration algorithm could provide means for software

migration, such that there is always a sufficient number of redundant copies of application software within the cabinet[10] (the software downloading bus should be used for that purpose). Clearly, reconfiguration should be considered more important and time-critical than software migration, and therefore the actual reconfiguration should precede the software migration phase. The number of copies of an application would depend on its criticality and the memory available within a cabinet. Reconfiguration schemes supporting software migration are likely to be more robust and provide a high level of fault tolerance for very rare failure conditions (multiple, non simultaneous failures of core modules), although they are also likely to be very complex.

> Commentary:
> Some problems could arise when a failure of a core LRM occurs before the software migration phase related to the previous failure had been completed. Software migration events related to different core LRMs would have to be suspended and queued waiting for completion of the reconfiguration of the avionics applications (actual reconfiguration should have the highest priority). Thus, the reconfiguration and software migration phases would have to be treated as separate events rather than a complex integrated process. This could impose additional difficulties with the proof of reconfiguration method determinism.

***Gateways***

As gateway modules are responsible for communication between the ARINC 629 global data bus and ARINC 659 backplane bus their failures can lead to very severe failure modes. As discussed in section 2.3.1.1 the fault tolerance related to gateways depends also on the way they are connected to the ARINC 629 data bus. In the preferable case (see Figure 2.7) the failure of any single gateway module does not affect the system performance, providing that at least one ARINC 629 global data bus and ARINC 659 backplane bus are still working.

Since gateways are used for transferring non-critical data as well as critical data, the failure condition leading to the loss of all gateway modules should be extremely improbable (less than one event in $10^9$ flight hours), as it would be followed by the loss of the whole cabinet. Gateways must be considered as safety-critical resources.

---

[10]The software migration process can be split into two phases: duplication and transfer. The first would create a copy of the required software, the second would transfer it to the destination module. Some problems with application

***Buses***

Both ARINC buses (global and backplane data bus) also constitute highly critical resources. While the failure of the backplane bus leads to the loss of a cabinet[11] (core modules do not support direct connection to the ARINC 629 data bus), the loss of the global data bus leads to the loss of all avionics systems and total system shutdown. Dedicated redundancy should be employed to achieve the necessary level of fault tolerance for those two buses.

Since core module design supports reconfiguration schemes without any need for dynamic in-flight software downloading, the software downloading bus can be considered optional and thus non critical. Even in the case of reconfiguration methods employing a software bus, the total loss of such does not necessarily lead to any system functionality degradation (this is subject to a subsequent core module failure), and only reduces system reconfigurability.

## 2.3.2. ARINC 651 example "D" based architecture

### 2.3.2.1. Design overview

The architecture shown in Figure 2.8 (below) follows the specification of ARINC 651 example "D". Most of the features are the same as in the original design, and the only required changes refer to the core module processing model and to the introduction of a dedicated software downloading data bus.

---

coherence could arise with this approach, if the copy being transferred was omitted by a state update action.

[11]Very interesting possibilities emerge from the presence of the software downloading bus presence in the cabinet. Since the backplane bus is more critical, in the case of its loss, the software downloading bus (or some of its redundant copies) could be dynamically switched into ARINC 659 bus mode, and thus be used as a backplane bus.

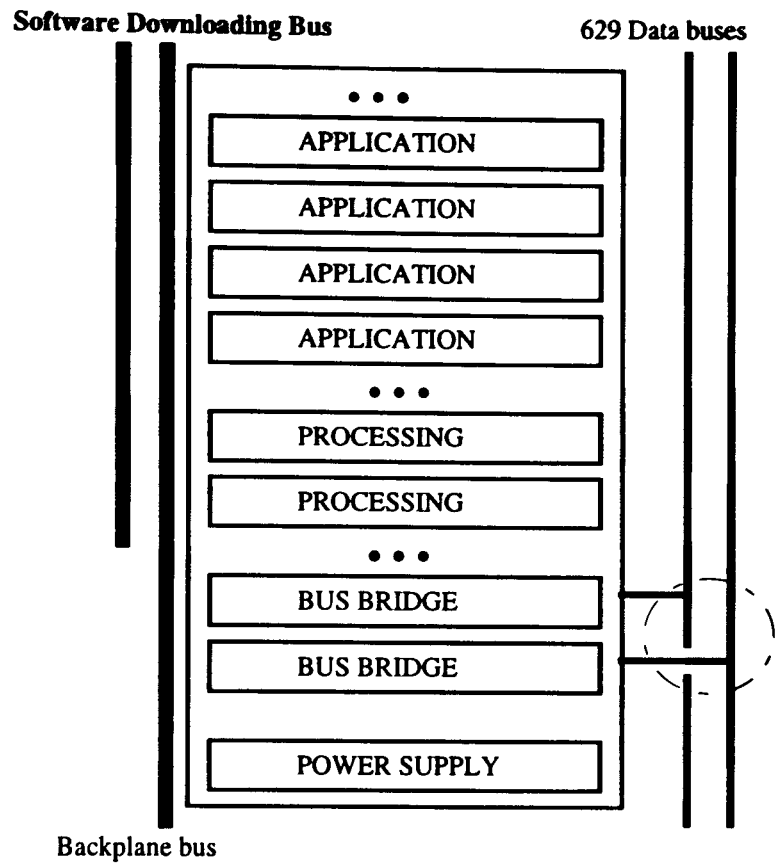



**Figure 2.8. Reconfigurable IMA based on ARINC 651 architecture "D".**

### *Modules*

Unlike in the previous example, core modules provide only processing power, and the responsibility for software storage and state updating lies with the application modules. Since the discussion on processing modules and reconfiguration from section 2.3.1.1 also holds in this case, the SESA model is preferable to the SEMA approach. Core module components follow the specification from section 6.7.1 of the ARINC 651 report with the addition of a software downloading data bus interface.

The application module design again follows the description from section 6.7.1 of ARINC 651 with the addition of a software downloading data bus interface. The responsibility of an application module is to store software for an avionics function or functions, and to update their states.

Similarly, bus bridge module functions are as stated in ARINC 651 section 6.7.3, and essentially provide an interface between the backplane and system data buses. The discussion on gateway module connection to the ARINC 629 global data buses from section 2.3.1.1 of this chapter also holds for bus bridges in this architecture, and a similar solution should be chosen to achieve the necessary level of fault tolerance.

***Buses***

The global system data bus and backplane bus are implemented as proposed in the ARINC 651 report ([2] section 6.7), although - as in the previous example - the actual specification of the backplane bus could change should a more attractive alternative emerge.

The software downloading bus should be implemented as a high throughput, possibly optical fibre based data bus, and its protocol should be optimised for the burst mode transfer with the purpose of shortening the downloading time. However, unlike in section 2.3.1.1 of this chapter, since core LRMs have no means for storing software, the software downloading bus is not optional, as no dynamic reconfiguration would be possible[12] should it not be available. This clearly follows the fact that the new application software has to be downloaded to a core LRM from an application module in order to successfully complete the reconfiguration process.

### 2.3.2.2. Points of failure

In this section various failure conditions related to failures of different system components are discussed. The analysis is qualitative and focused on the cabinet level.

***Core modules***

The discussion from section 2.3.1.2 (subsection "Core modules") of this chapter applies to this sub-section, with the difference that the core module in this RIMA architecture example is not responsible for software storage, so that multiple failures of core LRMs do not reduce software replication, and thus they do not reduce the level of reconfigurability. Moreover, there is clearly no need for software migration after a core LRM failure. When activated, the reconfiguration mechanism should employ the software downloading bus to transfer a copy of the required application software to the core LRM being reconfigured from an appropriate application module.

[12] As already mentioned in sections 2.3.1.1 and 2.3.1.2 the use of a backplane bus for software downloading purposes is not desirable.

***Application modules***

A failure of an application module decreases the number of replicated copies of some applications, but does not automatically induce any performance degradation. It does however, reduce the level of reconfigurability of the system, as fewer and fewer replicas of the necessary application software are available. In the case of multiple failures, that could lead to exclusion of some avionics functions from the reconfiguration scheme, should the software replicas already be exhausted.

To provide the necessary level of software replication, the reconfiguration mechanism could provide means for software migration, such that there are always redundant copies of application software. A dedicated software downloading bus should be used to support software migration. Note that unlike in RIMA architecture "C" (section 2.3.1), the software migration phase does not follow a failure of a core LRM but a failure of an application module. Therefore, in the case of RIMA architecture "D" the software migration and reconfiguration processes are distinct as they refer to failures of different modules. The software migration phase could be understood as lower priority reconfiguration of application modules.

However, in the case of both RIMA architecture proposals the software migration mechanism introduces extra functionality to the system, and the benefits of greater fault tolerance have to be verified against additional difficulties with the formal proof of the reconfiguration scheme resulting from increased system complexity.

***Bus bridges***

As the functions of bus bridge modules are analogous to those of gateways in the previous RIMA design, the discussion from section 2.3.1.2 of this chapter (sub-section "Gateways") applies here also. Both bus bridge and gateway related problems are of generic nature for proposed RIMA architecture examples.

***Buses***

The discussion on the ARINC 629 global data bus and the ARINC 659 backplane bus from section 2.3.1.2 of this chapter (sub-section "Buses") also holds for RIMA architecture example "D". However, different issues have to be addressed when the software downloading bus is being discussed.

Unlike in the previous example, the software downloading bus is not optional and it is necessary[13] for reconfiguration purposes. However, although the loss of the software downloading bus makes reconfiguration impossible, it does not immediately lead to system performance degradation. Only in the case of a subsequent core module failure would the loss of an avionics function be inevitable, provided that there is no dedicated redundancy in the cabinet. The software downloading bus in the example "D" has to be considered critical, and appropriate redundancy will be required to achieve the necessary level of fault tolerance.

### 2.3.3. Summary

The design drafts described in section 2.3.1 and 2.3.2 constitute possible implementations of reconfigurable IMA architecture. Both examples are closely based on solutions proposed in ARINC 651, and any changes required for reconfiguration have been minimised. In the most limited versions they could be restricted to some minor core module design changes and elimination of dedicated I/O modules for architecture example "C", and introduction of a software downloading bus and some core module changes in architecture "D".

Although the change of core LRM processing model from SEMA to SESA seems a relatively easy step (no additional functionality is required to accomplish this task, and in the worst case a SEMA capable unit will be forced to execute just a single application), the introduction of a specialised software downloading bus might prove relatively more difficult and costly, rendering architecture "C" as a possible preferred choice.

Despite the need for some changes, the two IMA architecture examples ("C" and "D") seem to be promising as bases for successful implementation of dynamically reconfigurable avionics systems.

---

[13] The use of the backplane bus for downloading software has been already identified as undesirable.

# Chapter 3. Review of Existing Reconfiguration Methods

## 3.1. Introduction

Dynamic reconfiguration schemes lead generally to a high degree of system integration, because core LRMs can be requested to perform functions from different avionics systems. Therefore it has to be considered as one of the most critical aspect of the RIMA cabinet, since a malfunction of the reconfiguration scheme can easily endanger the aircraft safety. In order to minimise the risk related to possible design or implementation errors in the reconfiguration scheme, the reconfiguration method should address various issues such as the equality and independence of processing nodes, determinism and integrity of the reconfiguration scheme, reliable communication between physically separated modules, maintenance of phase-synchronisation and data consistency, robust failure-detection and others. To avoid the possibility of introducing single points of failure, the reconfiguration schemes to be implemented in RIMA are also expected to operate autonomously, without any need for executive module or modules employed purely to control the reconfiguration process. A detailed discussion on requirements addressed towards reconfiguration schemes is given in Chapter 5.

Clearly, there can be many reconfiguration schemes satisfying such requirements, and thus it is necessary to develop schemes that exhibit the most desirable features with respect to different factors such as reconfiguration delays, determinism, simplicity, etc. Such schemes and their implementation can later be validated with the use of formal methods and/or testing.

In this research the following approach to the design of reconfiguration schemes was chosen. In the first stage, before any review of existing reconfiguration methods had been pursued, some initial schemes were designed and implemented using a software model of a RIMA system. These initial schemes are thus unlikely to be biased towards any existing solutions. In the second stage, a review of existing reconfiguration methods was conducted in order to identify solutions possibly applicable to RIMA systems. These solutions were then used in conjunction with the initial schemes in the third stage to guide the design of the most desirable reconfiguration algorithm (Chapter 6 and Chapter 7).

In this chapter the results of the second stage (literature search in the domain of reconfiguration and similar subjects) are presented. As the employment of reconfiguration for the purposes of sustaining the most critical functions of a system is a relatively new and application specific problem, not many publications were found that refer directly to this domain. In order to overcome the problem, the literature search was extended to include not only dynamic reconfiguration, but also task assignment and scheduling, resource allocation and particular implementations of distributed real-time systems. As most of the papers did not relate strictly to the problem of dynamic reconfiguration in RIMA systems, they were analysed with the purpose of establishing ideas that could be incorporated into the reconfiguration of avionics systems.

The remaining part of this chapter is organised as follows. In section 3.2 basic concepts related to the problems of reconfiguration, resource allocation, task assignment and scheduling are defined. The subsequent section (section 3.3) discusses various methods implementing the above ideas, and section 3.4 presents the conclusions of this phase of the research.

## 3.2. Concepts of reconfiguration

In this section notions related to the problems of reconfiguration, resource allocation, task assignment and scheduling are discussed. They are intended to give a better insight into those areas and allow better understanding of the following sections. Also, the relevance of each of the problems mentioned above to reconfigurable avionics systems is discussed.

Note that the discussion is focused on the level managed by the reconfiguration scheme, e.g. assignment of avionics functions to core LRMs. Thus, problems related to scheduling or allocation within a core LRM are not addressed, as they are transparent to the reconfiguration scheme and should be dealt with by the module operating system (OS) or application executive (APEX) [10].

### 3.2.1. Resource allocation

In the case where the resources necessary for computation of multiple tasks are strictly limited, the tasks compete in order to acquire the system resources. The idea of resource allocation can be simplified to a

problem of finding an assignment of system resources to computational tasks, such that the number of tasks that can be executed without conflicts is maximised. Various resource allocation methods are presented in [11], [12],[13],[14], [15] and [16], and are discussed further in this chapter.

In RIMA systems, processing modules as well as data buses can be considered as system resources shared by the avionics functions. Whilst the access to a backplane or a software downloading bus is solved by means of the data bus protocol, some provisions need to be made for allocation of core LRMs to particular avionics functions. The initial allocation is relatively straightforward as ideally there should be at least as many processing modules as there are avionics functions (or groups of functions) to perform. However, problems with resource allocation may arise when - due to failures of processing modules - there are not enough processors to compute all functions at the same time.

### 3.2.2. Task assignment

The problem of task assignment refers to the situation where tasks arrive in a multi-processor system and need to be assigned to processing nodes. As the computational capacity of processing nodes is clearly finite, in the case where many tasks arrive frequently at the system, the system processing capacity may become exhausted. Moreover, in many cases the algorithm needs to be designed with overall system load balancing in mind. Different task assignment methods are described in [17], [18], [19], [20], [21] and [22].

Avionics functions do not, technically speaking, arrive in a RIMA system during its operation, because the set of avionics functions performed within a cabinet is established during system integration phase. However, in the case where a critical avionics function is lost due to a module failure, the need for its restoration can be understood in terms of a task arrival, where a previously lost function arrives into a system as a new task.

Both task assignment and resource allocation methods refer to a similar class of problems encountered in multi-processor systems. However, they approach them from alternative points of view - assigning tasks to system resources against allocating resources to computational tasks.

### 3.2.3. Reconfiguration

Reconfiguration is widely used to limit adverse effects related to a failure of one or more processing nodes. It will usually include problems such as process reinstatement (which could also involve state transfer or state restoration), suspension of inter-process communication during reconfiguration, re-establishment of communication channels and transparency of process physical location for communication purposes.

In this chapter the notion of geometric reconfiguration [23] is of most interest. Geometric reconfiguration describes a wide class of situations where a part of a distributed application or an independent task changes its physical location from one processing node to another. The node can be understood as processor in a multi-processor system or a complete machine in a distributed system. Dynamic reconfiguration describes situations in which reconfiguration is performed during the system operation.

Various reconfiguration and recovery mechanisms are discussed in [16], [19], [23], [24], [25], and [26].

### 3.2.4. Scheduling

The notion of process scheduling refers to the problem of time slot allocation for multiple tasks on multiple processors. Certain levels of fault-tolerance can be achieved with scheduling techniques where more than one copy of a task is scheduled for computation on multiple processing nodes in the system, so that failures of processing nodes do not necessarily affect the deadline by which the task is completed.

Although, as mentioned above, this chapter does not deal with the low level process scheduling that should be dealt with by the processing module OS or APEX; however it is expected that certain scheduling techniques could be adapted for fault-tolerance purposes in RIMA systems.

Different solutions to the problem of fault-tolerant scheduling are presented in [16], [27], [28], and [29].

## 3.3. Literature review

Three main classes of approaching the problem of reconfiguration have been identified; these are architecture/connection orientated methods, communication based techniques and backup/replica process redundancy solutions (see sections 3.3.1, 3.3.2 and 3.3.3 respectively). Particular solutions may employ any combination of the techniques mentioned above, or may use alternative novel approaches (these are presented in section 3.3.4).

### 3.3.1. Architecture/connection based reconfiguration

Methods classified as architecture or connection based are particularly applicable to massively parallel computers, where the computer architecture (the network of inter-processor connections) determines system fault-tolerance. In such systems redundant (spare) processing nodes are used as backup for primary processors [24]. In the case of a primary processor failure, links between nodes are updated to allow a spare processor to take over the lost function. Clearly, the capacity for withstanding failures is a function of the number of redundant components and the flexibility of the inter-connecting network. Although the physical architecture of RIMA systems (see Chapter 2) is rather different from that of massively parallel computers, it can be expected that imposing a virtual architecture on top of the physical one could allow for mapping of the discussed methods into RIMA.

The authors of [24] assume that only those primary processors that are connected to at least one redundant component are repairable due to constraints of the interconnection network. They give two examples of fault-tolerant architectures that are presented in Figure 3.1 and Figure 3.2. The reconfiguration scheme attempts to switch the faulty primary processors to a spare one, to maximise the number of repairable components.

> Commentary:
> Consider the tree-like architecture as in the Figure 3.1. In the case of a failure of the processor "3", either spare processor "A" or "C" can take over the lost function. If the spare component "A" is switched in this situation, the processor "1" will no longer be repairable, as it will not be connected to any redundant one. On the other hand, if the spare processor "C" takes over, all other primary processors will still be repairable.

A bi-level reconfiguration technique is presented in [24], where the behaviour of the reconfiguration algorithm varies depending on the mode of operation. In the so called strict mode (where fast processor switching is enforced by the time constraints of a real-time application) the algorithm is sub-optimal, and it searches for a solution taking into account only locally available knowledge. In the relaxed mode (no critical time constraints) the algorithm searches globally to find the optimal switch sequence. Neither of these algorithms seems to be appropriate for RIMA systems, as search techniques are generally time consuming, and when used in combination with heuristics[14] may lead to reconfiguration non-determinism, which is strongly undesirable in safety-critical applications.

However, it seems possible to base a RIMA reconfiguration scheme on some sort of a virtual fault-tolerant architecture similar to those shown in Figure 3.1 and Figure 3.2, where spare nodes may represent redundant core LRMs or processing modules performing non-critical functions.

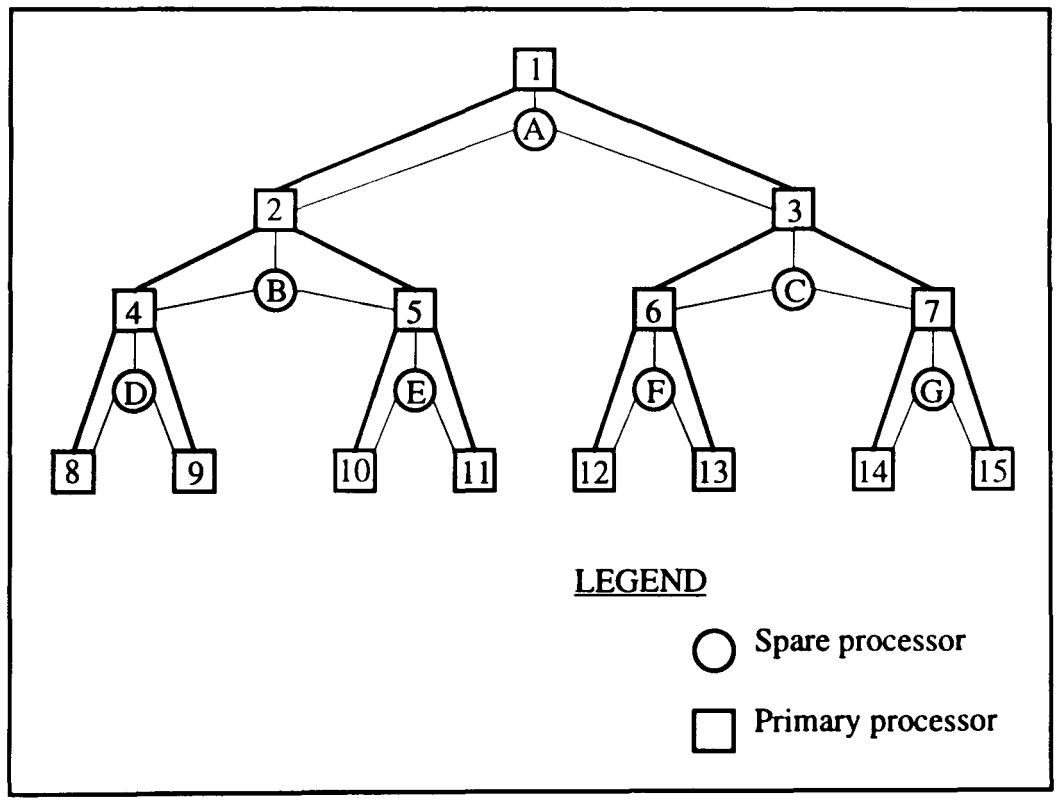



**Figure 3.1. An example of a tree-like fault-tolerant architecture.**

Consider the tree-like architecture shown in Figure 3.1, above. Processors "8" to "15" and processor "1" are connected to exactly one redundant component, and they could perform non-critical avionics functions. Processors "2" to "7" are each connected to two spare nodes, and thus they could run critical

[14] An employment of heuristic techniques constitutes a trade off between the search speed and the probability of

functions. In order to reduce the necessary system redundancy (six spare nodes are required in Figure 3.1 for fifteen primary processors), spare nodes connected to critical functions could correspond to modules executing non-critical functions. For example modules "12" and "C" could be implemented by the same physical device.

It is expected that architectures such as the redundant binary tree (Figure 3.1) or the augmented mesh (Figure 3.2) could be successfully employed for implementation of reconfiguration schemes in RIMA systems, although they may lead to sub-optimal reconfiguration chains.

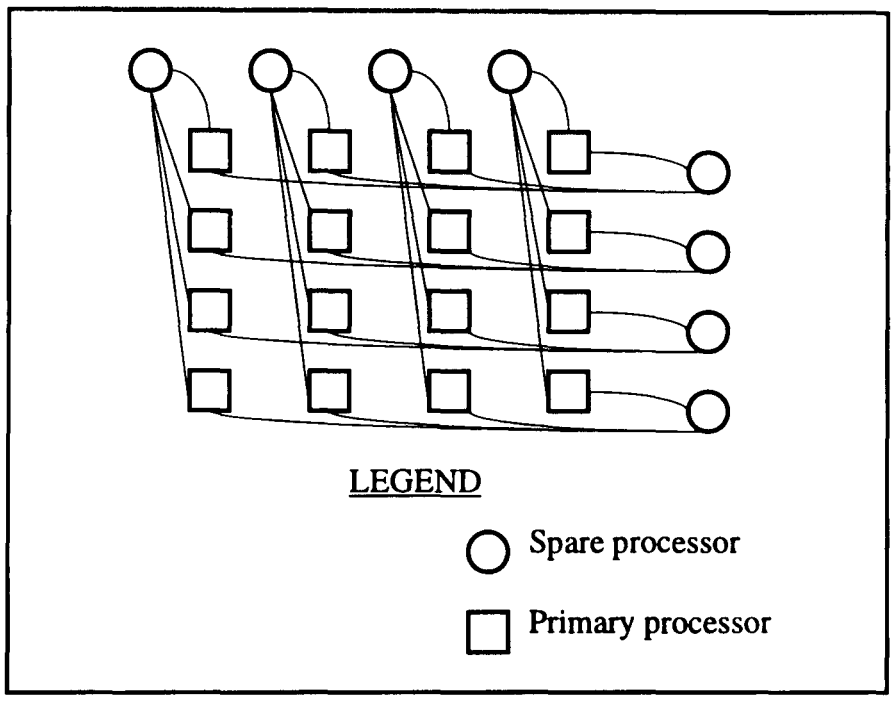



**Figure 3.2. An example of a fault-tolerant augmented mesh architecture.**

In [12] and [14] the authors consider different algorithms for resource allocation in a multi-component computer system. The bit reversal, buddy, grey code and free-list strategies are identified as possible solutions.

The first three of the aforementioned methods name all the system components in the strategy specific manner (the component name is represented as a number corresponding to the node location in the system). The resource allocation strategy keeps track of the availability of the named components, maintaining a binary word describing the system state (where particular bits represent appropriate system nodes, where 1 means that the node has already been allocated, and 0 means the node is still available).

---

finding the optimal solution.

Since in RIMA a non-critical but allocated node may still be required to reconfigure in order to sustain a critical function, these techniques do not seem to be applicable for reconfiguration of RIMA cabinets. However, some strategies for module naming can be embedded into RIMA in order to allow simple identification of particular LRMs.

The free-list approach employs a list of available system resources (the authors apply this solution to hypercube computers where available resources represent non-allocated sub-cubes). When a new task arrives in the system it is allocated the necessary resources, and the list is updated. When a task terminates, its resources are reclaimed by the system, and the free-list is updated again.

In the case of RIMA systems multiple resource lists could be employed that would represent processing modules available to particular avionics functions (the lists could include redundant core LRMs as well as the processing modules executing non-critical functions). When a function is lost due to a module failure, it is treated as a new task arriving at the system and it will be allocated a core LRM from its free-list, unless the list is empty for example due to resource exhaustion. If a module is allocated to function "A", it would be removed from the free-lists of all functions of the same or lower criticality, as only the loss of a more critical application can lead to further reallocation.

In [20] the authors suggest the use of a hierarchical queue for the assignment of tasks to processing nodes (see Figure 3.3). A new task is placed in the root queue and then it is passed down the tree until it reaches an available processor (called a leave node). The process of tasks descending through the hierarchical queue can be ruled by factors such as the number of tasks already queued under a branch, available capacity in the child queue, overflow of the parent queue and others.

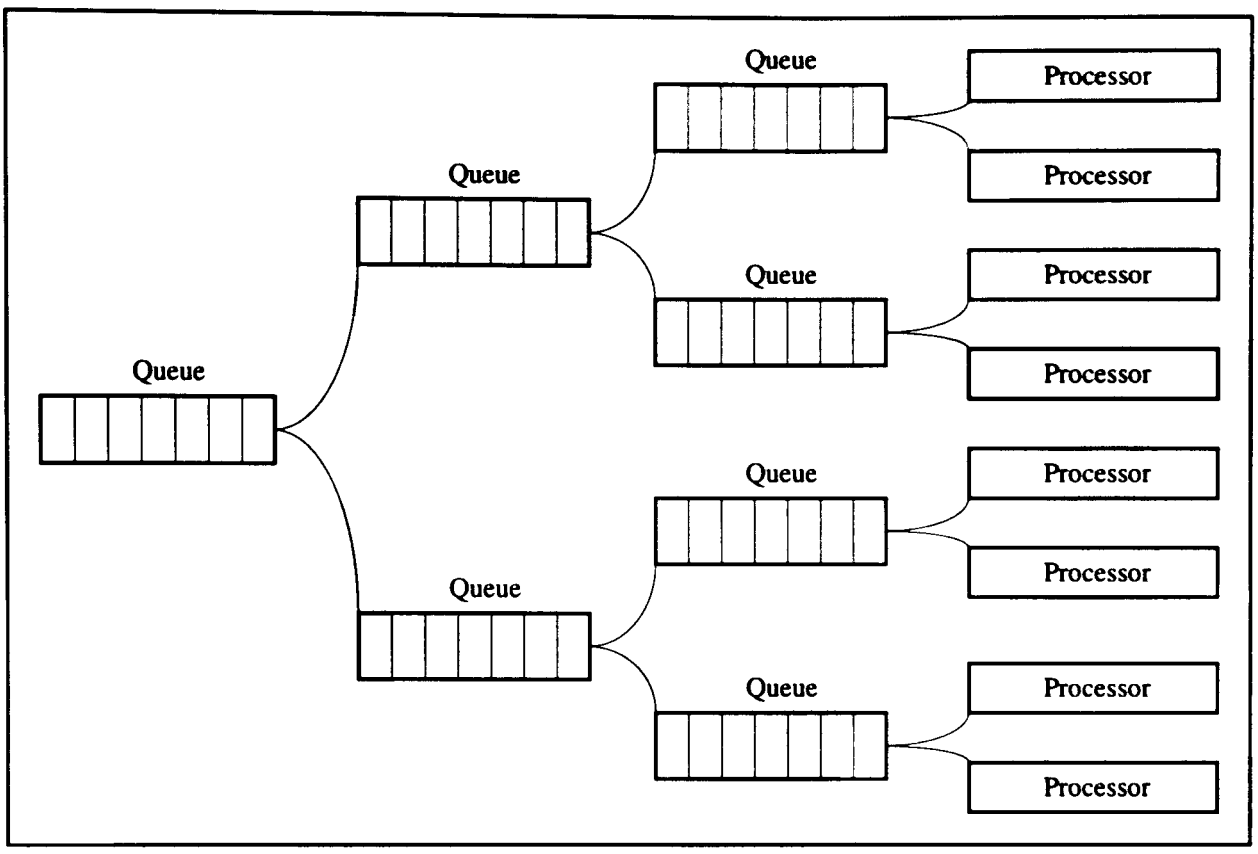



**Figure 3.3. Hierarchical queue organisation for task management.**

The applicability of a hierarchical queue organisation to RIMA systems is rather limited. Although it is possible to place the avionics function that needs to be restored in the root queue and then allow it to progress through the system, the real strength of such system organisation shows only in applications where multiple tasks arrive relatively frequently. Another problem associated with the use of the queues is their "First In First Out" (FIFO) service policy, that would allow a non-critical task to "block" a queue while waiting for an available core LRM[15]. It is unlikely that queues or queue-like solutions can be employed for the purposes of autonomous dynamic reconfiguration in RIMA systems.

### 3.3.2. Communication based reconfiguration

Communication-based reconfiguration schemes rely on the module capacity for message exchange. Communication channels can be used for purposes such as scheme synchronisation, event signalling and verification, and finally software or data exchange. Communication based reconfiguration schemes exhibit a great potential for highly robust behaviour, as transient faults of a single processing module or

[15] It is possible to design prioritised queues, where more critical tasks can be served first; however, it would lead to an additional increase of software complexity.

faults of a relatively small group of modules can be masked or neutralised by the means of majority voting or a similar technique.

Mechanisms for message exchange employed by particular systems may differ significantly with respect to both the physical connection model and the hardware/software protocols. Therefore, reconfiguration schemes may vary considerably between systems, and solutions implemented in one system may not be applicable to another.

In RIMA cabinets, all processing modules are connected to a backplane bus that can be used for message exchanging, and thus the communication path between any two core LRMs is of the length of one. Such a connection model allows the RIMA system to emulate architectures of any desired length of the communication path.

> Commentary:
> It is a relatively simple task to implement longer communication paths in a direct connection system, as a message can be passed via multiple modules on its way to the required destination. On the other hand, it would be rather difficult to implement a direct one-to-one communication model in some inherently indirect systems such as tree-like or queue-like architectures (in such architectures messages may need to be re-transmitted by multiple nodes before they reach their destination).

Some problems can be encountered if a reconfiguration scheme requires multiple modules to conduct communication tasks simultaneously via independent connections. The ARINC 659 data bus standard was designed as a multiple source serial communication medium, and does not allow for broadcasting by multiple sources at the same time. It could be possible to overcome this problem partially by appropriate time slicing with the assumption that two messages exchanged within a specified time interval are simultaneous.

A strongly communication-based approach to load balancing via process migration in a distributed system (called the Charlotte system) is presented in [18]. In the Charlotte system particular processes may have to be migrated from one site to another in order to obtain better utilisation of system resources. The migration algorithm is based on negotiations between multiple processing nodes. In order to find a new location for a particular process, the source (the processing node that was executing the process prior to reconfiguration) sends an appropriate message into the system indicating the need for a process transfer.

The actual migration commences when a confirming message arrives from a destination site (i.e. a processing node indicates its willingness to perform the task). In such a situation the communication to and from the process being transferred is blocked (processes attempting communication with the one being transferred are suspended) and the messages are queued. The task state is transferred to the new site (including all data, stack, register values and variables), the process execution is restored and its communications are re-activated.

Although this approach in general seems to be unsuitable for reconfiguration of RIMA systems due to certain problems such as:

- a source node has to initiate process migration, which in the case of RIMA systems would have to be a faulty processing module,
- a transfer of the full process state would introduce a considerable communication overhead,

it could also constitute an interesting approach to preservation of software replication (because the appropriate number of spare software copies could be maintained via software migration utilities), and to the implementation of transparent process location for communication purposes.

In the Charlotte system, processes can communicate regardless of their physical location, i.e. after a migration from one processing node to another, all processes are able to continue message exchange without any changes to message addressing because the address of a process does not depend on its physical location. Such message addressing transparency would be very desirable in RIMA systems, where avionics functions can be performed by any of the core LRMs, and communication between processes will be required.

A process migration mechanism based on the Charlotte system could be employed into RIMA for the purposes of preservation of software replication. Clearly, there need to be multiple copies of every avionics function that may have to be restored after a module failure. However, in systems implementing process or software migration, the number of software copies stored in the cabinet at any point of time can be significantly reduced. Various issues related to software migration are discussed in Chapter 2.

An alternative solution to the problem of process state transfer is proposed in [23]. In this approach it is the underlying operating system that provides appropriate primitives for state abstraction. Also the services related to suspension of the process communication activities, removal of the existing communication channels and establishment of new connections are provided on the level of system calls. Such an approach allows implementation of very general reconfiguration models including reconfiguration of module implementation (a change of a part of an executing application to a newer or more accurate version), reconfiguration of structure (introduction or removal of particular software modules) and reconfiguration of geometry (a change to the physical location of a process). However, it is unlikely that operating system based solutions will be suitable for RIMA systems, and thus such an approach is rather impractical in the context of avionics systems.

> Commentary:
> In order to implement such a reconfiguration model it would be necessary to redefine the application executive (APEX) standards [10], as well as designing and developing a new safety-critical operating system supporting dynamic reconfiguration primitives.

The need for the development of a new operating system could be avoided if a solution similar to that of a Software-Implemented Parallel-System Fault-tolerant layer (SPF) as described in [30], was implemented in RIMA. The SPF layer exists between the operating system and the user application, providing calls to fault-tolerant services. At the compilation time traditional system calls are linked to their fault-tolerant versions in SPF libraries, so that appropriate services can be provided by the SPF during the application execution. However, it can be expected that the development cost of such a fault-tolerant and safety-critical software layer will be comparable with that of a fault-tolerant safety-critical operating system; thus no actual cost reduction will follow this solution. Moreover, introduction of an additional software layer into the system could inflict some new software related problems and would in general lead to an undesirable increase of system complexity.

In [25] a different approach to state preservation for the purposes of process re-execution is presented that avoids extensive interaction with the operating system or other software. The authors employ techniques for checkpoint saving, rollback and task restarting in order to implement a fault-tolerant system. At specific time intervals all the essential information related to the execution of a particular function is saved as a checkpoint. In the case of a failure, the process is rolled back to the last saved checkpoint and

then it is restarted. Such an approach will in general lead to some loss of synchronisation or some transient problems when a function is restored. These problems are clearly dependent on the frequency of checkpoint saving and the application itself. Problems related to restoration of execution of functions requiring state are generic to all reconfiguration schemes, and issues such as automated identification of application state, time intervals between checkpoints and problems related to transient discontinuity of function results require further discussion which is beyond the scope of this chapter.

Issues related to the problem of a rollback of multiple functions - as identified in [25] - also need to be addressed when RIMA systems are being considered. In the situation where some functions depend on results from other functions, a rollback of a data receiver may result in a rollback of the data source (a receiving function may need the data from the period of time between the last checkpoint and the failure in order to be able to re-execute). A log of messages received by the function between checkpoints can be used to avoid such undesirable situations as the restarted function can then process the data based on the message log.

In [13] the problem of task assignment in a distributed system consisting of homogenous processes communicating via a data bus is discussed. A resource allocation algorithm called MULTIFIT is presented, which appears to be adaptable to RIMA systems.

In MULTIFIT, an arriving task is assigned to a processing node (or alternatively, a processor is allocated to the task) based on an allocation strategy. The algorithms discussed in [13] implement two phases - sorting and assignment. In the first phase, arriving tasks are sorted based on some required criteria (in RIMA it could be their criticality), and in the second phase they are assigned to appropriate processing nodes. As an arriving task in RIMA would generally represent a previously lost function, the sorting phase could possibly occur during system initialisation or in the case of simultaneous multiple failures. In the first case the whole set of functions can easily be pre-sorted during system integration and thus no real-time computational overhead would be encountered, and the second case should be extremely improbable[16], and thus it would not have to be dealt with.

[16] An appropriate qualitative and quantitative analysis is conducted Chapter 8.

To implement the assignment phase some sort of a resource list (possibly similar to those of [12] and [14]) would be required. Based on the allocation strategy (e.g. best fit or first fit) the algorithm would search through the list of processing modules to find an appropriate one. The best fit strategy would involve searching for the least critical module that stores the appropriate software in its non-volatile memory, and thus is able to take over the lost function. The first fit would assign the function to the first module found that stores all the required software and whose own application is less critical than the lost function.

Another way of selecting tasks for assignment to processing nodes is proposed in [15], where clients (functions, tasks) have to place requests for resources. The system allocates available resources to whichever client has an eligible request and the earliest virtual deadline. The authors define the virtual deadline in terms of virtual time units, the length of which depends on the number of performed tasks and their criticality.

A similar approach could be used in RIMA to implemented module recovery. Each function that was previously lost and needs to be restored would represent a client with an eligible request for resources; the most critical of the clients could be understood as that with the shortest virtual deadline. Thus when some resources become available in the system (e.g. a processing module had recovered from a transient fault), they would be allocated to this function. However, such a model is not directly applicable to reconfiguration of RIMA, as a temporarily lost critical function has to be allocated some resources even if none is immediately available (a less critical function needs to be sacrificed).

A strictly non-autonomous method for resource allocation in distributed systems is described in [11], and although its seems to be unfeasible to adapt it to an autonomous RIMA cabinet, certain features of this solution could be of some interest where reconfigurable avionics systems are concerned.

The authors discuss a distributed system where global allocators (GA), local allocators (LA) and local schedulers (LS) co-operate in the process of resource allocation. Based on its general knowledge about the system load, a global allocator sends the tasks to LAs based on local sites. Local allocators maintain detailed information about the locally available resources, and they are responsible for passing the task to

one of the local schedulers. Finally, the LS either accepts the task for scheduling or rejects it and sends it to the next recipient based on focused addressing. Although non-autonomous and non-applicable to reconfiguration of a RIMA cabinet, such an approach could be considered for adaptation to some global reconfiguration scheme. In such a solution functions can be reconfigured between separate cabinets, and some mechanisms may have to be provided in order to determine which cabinet and which module within the cabinet needs to reconfigure in the event of a function loss. However, as global reconfiguration schemes seem to be highly unfeasible (see discussion in Chapter 1), this technique is of little interest.

It is the notion of focused addressing that can be used for reconfiguration in RIMA systems. In the focused addressing mode every message has its destination address as well as the addresses of further recipients. If the first destination rejects the message - which in RIMA could represent an avionics functions - it is passed to the next one in the list. Similarly, in a RIMA cabinet a function that needs to be restored could be passed from one core LRM to another until a suitable one is found (that would involve matching factors such as criticality and software availability). The generation of the addressing list for each function could be done statically during system integration, which would lead to faster reconfiguration (since there would be no need for dynamic list generation), and would allow easier verification of list integrity constraints.

Although none of the approaches discussed above implements thoroughly a dynamic reconfiguration scheme, particular features such as focused addressing [11], process migration [18] or checkpointing [25] can be implemented into RIMA systems in order to achieve robust fault-tolerant behaviour.

### 3.3.3. Backup/replica based approaches

A class of fault-tolerant techniques where a secondary copy of the tasks is placed in the system along with the primary task, is discussed in this section. While backup will usually refer to an inactive (non-executing) copy of a function, the notion of replicas embraces both active and inactive tasks. Moreover, systems may also employ semi-active replicas, that can be understood as functions that process their inputs but do not produce outputs or their outputs are ignored[17].

[17] Similar solutions are widely used in avionics systems to provide redundancy of particular avionics functions.

Variations on the primary/backup (PB) technique applied to fault-tolerant scheduling of tasks in a multiprocessor system are discussed in [27], [28] and [29]. The PB approach depends on the scheduling of a single backup copy of a task on a processor different from that executing the primary task. Thus, in the event of a failure of the primary processor, the backup copy can be executed in order to complete the function before its deadline.

> Commentary:
> To be able to complete the task before its deadline in the event of a failure, it is necessary that the task time window is at least twice as long as the worst case task execution time. This is required in order to guarantee the task completion in the case when both the primary and the backup copy have to be fully executed due to the faulty results of the primary task. Note that primary and backup are not executed simultaneously in order to avoid computational overhead in the case of the correct execution of the primary task.

In [27] the authors extend the traditional PB scheme by introducing the notions of backup overloading and backup deallocation. In backup overloading multiple backup copies of different processes are scheduled for the same processor and overlapping time slices, so that the total time reserved for the scheduling of backup copies is reduced. When the primary task finishes successfully its backup copies are deallocated in order to provide additional computational time in the system. However, this approach leads to certain problems, as only a single processor failure can be tolerated within a time interval.

> Commentary:
> If two or more processors fail within a short time interval, multiple backups have to be activated. However, as time slices reserved for those backup copies overlap, it will not be possible to complete all tasks within the time constraints. Some functional degradation of the system will follow such a scenario of events.

Issues of scheduling primary and backup tasks on multiple processors with the intention of minimising the time lost due maintaining system fault-tolerance are further discussed in [27], [28], and in [29] the scheduling techniques are enhanced to tolerate transient faults. However, as they do not guarantee a schedule feasibility for a given set of tasks, they are unlikely to be adapted for reconfiguration of RIMA systems.

An example of an autonomous dynamic reconfiguration method - based on a single backup PB approach[18] and applied to a RIMA system - is presented in [31]. Various reconfiguration schemes similar to the algorithm presented in [31] were independently designed and implemented in the first stage of this phase of the research (see section 3.1) as initial reconfiguration schemes. However, they differ from the traditional primary/backup schemes as multiple backup copies are employed, and they are assigned to particular processing modules rather than scheduled. Variations of these are discussed in Chapter 6.

The approach to distributed fault-tolerance in the Delta-4 system presented in [19] employs multiple process replicas in order to provide redundancy. Objects called capsules (generally representing processes), are replicated across multiple physical machines to create fault containment regions and reconfiguration domains. Three types of replicas (active, passive and semi-active) can be identified in the Delta-4 system. In the case of a failure of a leader replica (similar to a failure of a primary task in PB), a secondary replica takes over the execution and a new replica is created to sustain redundancy. This is a somewhat alternative approach to software replication maintenance to the one presented in [18] (see section 3.3.2).

Backup/replica approaches seem to address closely many problems related to reconfiguration in RIMA systems. Therefore, it is expected that reconfiguration schemes implemented in RIMA may follow certain solutions presented in this section. However, it is unlikely that a pure backup/replica approach will be able to deal with all issues related to dynamic autonomous reconfiguration of avionics systems, and thus it will have to be refined with additional techniques related, for example, to core LRM recovery from a transient fault or software replication maintenance.

### 3.3.4. Alternative approaches

In this section two seemingly deterministic approaches to reconfiguration are discussed. The first of them, based on the concept of a state machine, is presented in [26], the second - employing scenarios for specification of the set of functions performed by the system - in [16].

---

[18] Although the reconfiguration scheme is based on a single backup approach (where only one inactive standby copy of each function is resident in the memory of a processing module), the author makes provision for further backup in terms of "cold" standby copies that can be downloaded from a storage device if necessary.

In the state machine approach activities such as state saving, process migration, and execution restoration are modelled as sequences of transitions between different states of the system. Two types of transitions are identified: process changing transitions (transitions that can add or remove a process) and constraint changing transitions (transitions that modify permissible behaviour of a fixed set of processes). Reconfiguration of the system happens along a reconfiguration path that is defined as a sequence of reconfiguration and recovery transitions. Reconfiguration paths can be automatically calculated based on the dependencies between the old and the new state and on the reconfiguration constraints (a graph generated from the old and new configurations is used for path computation[19]). In this approach, reconfiguration of RIMA systems can be modelled as restructuring (modification of an application by adding or removing basic machines[20]), and software replication maintenance can be viewed as relocation (migration of a process from one processor to another).

As the allowed states of the system and allowed transitions can easily be constrained, reconfiguration schemes based on this approach could be designed to be strongly deterministic. Moreover, if a more strict state machine model, e.g. Petri nets, was to be used, the behaviour of the system would be strictly predictable. However, some problems can be expected when the data structures necessary for such reconfiguration (state graphs) and the computation necessary for finding reconfiguration paths are being considered. Although possibly deterministic, such an approach is likely to exhibit reconfiguration delays and reconfiguration data size deficiencies.

Commentary:
The reconfiguration data required to implement the state machine approach needs to accommodate information about all system states and state transitions. The number of possible states can be very high, and for n modules able to perform one of k functions each, the number of states could be as big as $k^n$ (or even $(k+1)^n$ to include faults). For a typical value of n equal to ten and k averaging around four, the number of possible states can be estimated as $5^{10}$. In such a situation the amount of required reconfiguration data could invalidate the feasibility of state machine based methods.

---

[19] Alternatively, the system designers could specify the new configuration and the system would derive all necessary transitions [26].

[20] The notion of a basic machine describes executable entities such as processes or capsules in [19].

In the second approach [16], scenarios are used to provide fault-tolerance based on reconfiguration, load shedding and graceful degradation. A scenario describes the set of applications which have to be simultaneously executed by the system. When some system resources vanish (e.g. due to a failure) or there is a need for execution of a different application (e.g. in a military aircraft air combat functions are not necessary during take-off or landing), the system changes its execution scenario.

Load shedding is represented simply by a removal of an application from the scenario[21]. A change from an accurate version of an application to some less accurate copy, usually with smaller processing requirements, implements graceful degradation. Finally, in reconfiguration applications are removed from processing nodes and are redistributed onto a new set of available resources (e.g. failed processors are eliminated). Assignment constraints can be imposed in the reallocation phase to enforce allocation of appropriate applications to particular processors.

Although scenarios as described in [16] do not encapsulate any information about the assignment of tasks to resources (they only contain information about the set of applications to be performed), they could be easily refined to accommodate this need. In such a case, each scenario would deterministically describe the assignment of functions to processing modules, and scenario changes would correspond to consecutive events of failure or recovery.

> Commentary:
> Each scenario could also include information about required scenario changes that relate to particular failure or recovery events. For instance, if module "A" fails in scenario $S_1$ it will be changed to scenario $S_5$, if module "B" fails in scenario $S_1$ it will be changed to scenario $S_2$, etc. (see Figure 3.4). As all scenarios would contain all information about assignment of functions to processing modules (e.g. module A - function $F_1$, module B - function $F_4$, etc.) the system behaviour would be deterministic.

[21] The removal is actually implemented as a change from one scenario to another, that does not include the application being removed.

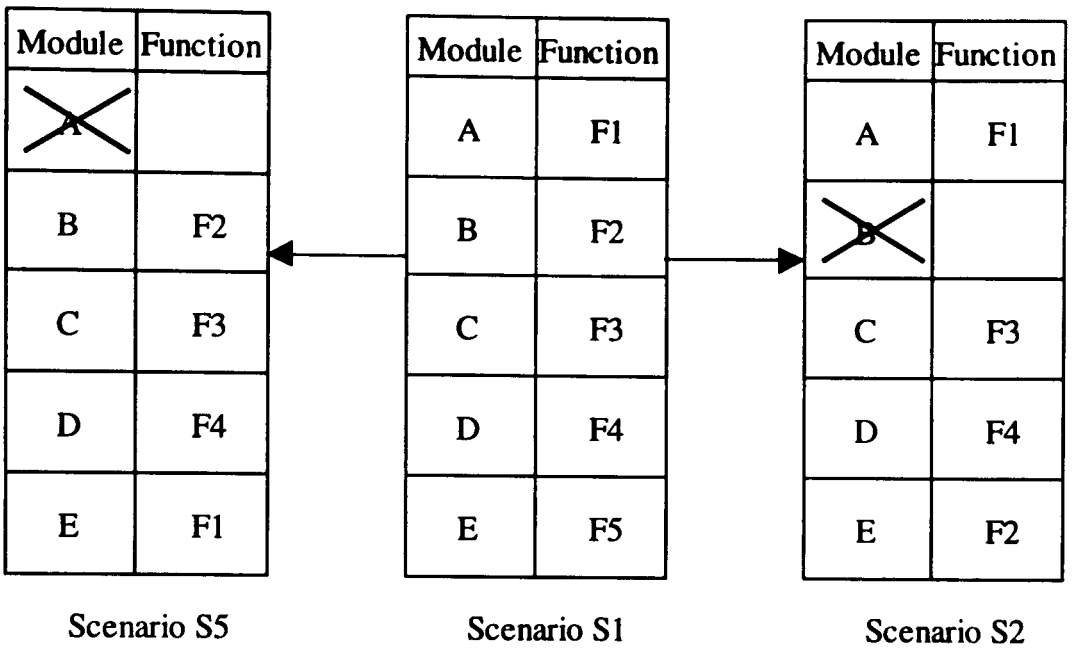



**Figure 3.4. Example of scenario changes.**

Some problems may occur, when the generation and storage of reconfiguration data specific to this approach are considered. It is likely that such data will be very complex, as specification of many scenarios and scenario changes corresponding to multiple failures and recoveries may be required, and a situation similar to that discussed for the state machine approach can arise. Complex and comprehensive reconfiguration data can be expected to be difficult to generate and prove consistent. Therefore, unless a specific implementation of this method allows for reduction of the data size, it seems to be rather impractical.

It needs to be said, that both approaches bring the benefits of deterministic and possibly autonomous behaviour (since each module could undertake its own decisions based on an identical state machine model or scenario data), at the cost of extensive reconfiguration data and/or the need for additional computation. Although they could be relatively easily adapted for reconfiguration of RIMA systems, due to their inherent time or storage inefficiency, the practical applicability of the aforementioned techniques is rather questionable.

## 3.4. Conclusions

In the previous section various reconfiguration methods and their adaptability for the purposes of dynamic reconfiguration in Reconfigurable Integrated Modular Avionics were discussed. As most of

them do not make any provisions for autonomous operation, their immediate applicability to RIMA systems must be doubtful. In general, if the algorithm depends in a deterministic manner on some reconfiguration data (e.g. the scenarios and scenario changes in section 3.4), such information can be replicated over all the processing modules in the cabinet. The determinism of the algorithm guarantees that all modules will undertake identical decisions based on the same reconfiguration data. Thus, a wide class of discussed methods can be modified to exhibit autonomous behaviour, although there are additional concerns regarding the possibility of data inconsistencies between modules.

Furthermore, if a scheme depends on some executive modules (e.g. task managers or resource allocators), each of the core LRMs can be given such a function. Again, providing that the algorithm behaves deterministically and the modules are able to accurately interpret the state of the system, all processing modules would undertake matching actions. However, some problems may occur if multiple modules decide to assign multiple copies of a single function to a single processing module. To avoid such problems, each of the processing modules would be assigned an influence domain, upon which it would be able to impose its decision. Clearly, if the domain is restricted only to the processing module itself, a mapping from a non-autonomous to an autonomous scheme is achieved.

In most of the general cases, it is possible to map a non-autonomous algorithm into an autonomous one, employing replication of data and authority, data consistency maintenance techniques, thus guaranteeing the undertaking of deterministic and consistent decisions, provided that all core LRMs are able to consistently monitor the state of the cabinet.

It seems very likely that some variations on the Primary/Backup approach will be employed to provide fault-tolerance in RIMA cabinets without a software downloading bus. However, the discussion also suggests that additional mechanisms may have to be implemented into the scheme in order to support maintenance of reconfiguration data consistency or software replication. A communication based algorithm is likely to be designed for these purposes. In cabinets with a software downloading bus, it is possible to design reconfiguration schemes without any backup or replica assignment, as the software can be provided dynamically. Also, it is expected that the most robust reconfiguration algorithms would have to be based on communication between core LRMs in order to reduce the risk of undertaking inconsistent

decisions due to misunderstanding of the cabinet state. However, it is also likely that such algorithms will be exposed to possible design or implementation errors due to their increased complexity.

It has to be concluded that - despite a wide literature search - no autonomous dynamic reconfiguration algorithms were found at the time, that could be directly implemented into RIMA systems, although parts of particular solutions could be used as basis for the design of appropriate reconfiguration algorithms.

# Chapter 4. Analysis of Configuration and Redundancy Requirements

## 4.1. Introduction

As already discussed in Chapter 1, RIMA systems are strongly based on IMA with the additional capability of dynamic in-flight reconfiguration. The RIMA systems are expected to be even more attractive with respect to their cost, although a question arises whether the new systems can be as reliable as the traditional ones and what would be the availability of such systems.

In this chapter the availability and reliability of non-reconfigurable and reconfigurable avionics are assessed with the use of the Markov state space analysis (details of this particular application of Markov analysis can be found in [32], whereas a discussion of the Markov approach can be found – for example - in [33], [34] and [35]). Although the assessment is focused on the reliability, availability and redundancy of processing modules (black boxes and core LRMs), some consideration is also given to other system components.

The aim of this chapter is to investigate three different types of avionics systems (black boxes - e.g. A320, IMA - e.g. some systems on Boeing 777, and RIMA - under development) in order to determine the safety and redundancy figures related to each system type. The comparison of such figures is expected to give a good insight into the possible attractiveness of the new approach.

The results presented in this chapter had been obtained from a tool designed and developed specifically for this purpose. A detailed discussion of the Markov method, related design and implementation issues and the actual software solution can be found in [32].

## 4.2. Processing modules redundancy

In this section Reconfigurable IMA systems are analysed in order to establish and assess their possible benefits, and an analysis is conducted in order to find the optimal parameters for a RIMA design. Also, a simple comparison between traditional (non-reconfigurable) approaches and RIMA systems is given.

As the size of RIMA cabinets[22] determines the number of avionics functions performed within a cabinet, it is a crucial factor that has to be taken into account while designing a RIMA architecture. Clearly in local reconfiguration schemes it is the number of modules (functions) in the rack that constrains the cabinet reconfigurability, and implicitly the reconfigurability of the whole system. It is expected that the capacity for dynamic reconfiguration will lead to a decrease of the required number of redundant core LRMs, thus it could lead to a decrease of the total cost of the avionics system.

To have a fair comparison between RIMA and other systems, a similar study of necessary redundant core LRMs was conducted for avionics systems that do not employ reconfiguration techniques. In both studies only the availability and reliability of processing modules was taken into account, and further considerations would be required to assess the influence of other system components such as power supply modules, gateways or data buses.

### 4.2.1. RIMA systems

In this study the unavailability factor of not more than 1% in 400 hours was required from avionics systems. This gives a 99% chance of being able to dispatch the aircraft after 400 hours of operation, and still meet all safety requirements not having required any maintenance during that period. The number of redundant core LRMs required to meet the safety [3] and availability objectives in systems implementing various numbers of avionics functions and employing various numbers of processing units per cabinet was assessed with the use of the Markov state space analysis [32].

The study was conducted for a range of cabinet sizes of 6, 7, 8, 10, 12, 14 and 16 core LRMs, and system sizes[23] varying from about 40 to about 90 avionics functions. The very small cabinet sizes of six and seven processing units were tested only for availability figures in order to establish whether cabinets of such sizes could work with just a single redundant core LRM (sizes of eight and more require at least two redundant modules). Since such cabinet configurations proved to be highly unavailable, the choice of cabinet sizes of seven or less is of little practical interest. The option of 90 avionics functions being

[22] The notion of cabinet size should be understood as the number of core LRMs in the cabinet.

requested from the system seems to be a little excessive as current systems implement only between about 50 and 60 avionics functions. However, such a high number has been chosen as an upper limit to leave some capacity for further growth of the systems.

> Commentary:
> The analysis was based on single application core LRMs, and thus it assumed direct equivalence between avionics functions and processing modules. In case of multi-application processing modules the same analysis remains valid provided that the whole functional groups (groups of functions performed by a single core LRM) are reconfigured together. Such functional groups can be treated as complex avionics functions, and thus they would preserve the mapping of avionics functions to core LRMs. With this approach a system of 80 prime avionics functions could be represented by a system of 50 functional groups and thus the analysis conducted here for systems of 50 functions would be sufficient for that case.

To obtain accurate availability and reliability figures for cabinets, the analysis was conducted in two phases

- cabinet level availability study
- cabinet level reliability study.

In the first phase the Markov analysis was performed with a time interval of 400 hours for cabinets consisting of processing modules with 10,000, 20,000 and 30,000 hours MTBF. The choice of the MTBF factors was based on the expectation that the RIMA modules MTBF would be around 20,000 hours[24]. The additional analysis for ten and thirty thousand hours MTBF was conducted in order to give a better view into the problem, as well as making the results valid for MTBFs varying from the projected one. The analysis of the first phase gave the probabilities of all possible failure combinations within the time interval. To achieve a reasonably low level of unavailability a cabinet needs to be able to tolerate certain modes of failure, i.e. after the 400 hour period the safety requirements for the next flight hour need to be met when dispatching an aircraft with a cabinet containing failed processing modules.

---

[23] The notion of system size should be understood as the number of avionics functions implemented by the system.

[24] This followed the discussions with the members of the aerospace industry, in this case British Aerospace AIRBUS Ltd.

In order to assess the core LRM redundancy necessary to meet the safety objectives (phase two), the Markov analysis was run for a one hour time interval[25] for different configurations of cabinets assessed in phase one. The analysis was run for cabinets that, at the start of a one hour flight contained varying numbers of failed core LRMs. For example, when starting with two failed modules, a probability of failure of $10^{-5}$ per flight hour (major mode of failure allowed probability) or less was achieved for another two failures, thus the system would require four core LRMs to implement a major function if no means for dynamic reconfiguration were provided. The influence of system capacity for dynamic in-flight reconfiguration was later assessed and it is discussed in section 4.2.2.

The results of phases one and two were combined in order to obtain redundancy figures for the whole system with the required level of dispatch availability. These were further used to assess the total number of core modules and the total number of redundant core modules necessary to implement avionics systems of varying sizes. A more detailed discussion on redundancy assessment is given later in this chapter, where the cabinet configuration study is presented in section 4.2.2. At this stage the discussion of results is mainly focused on the global (system) level, although some consideration is given to cabinets and their preferred sizes.

The following figures show the total number of processing modules (Figure 4.1) and the number of redundant core LRMs (Figure 4.2) required to implement avionics systems of sizes varying between 43 and 91 functions as a function of the cabinet size. The examination of results should indicate which cabinet sizes give the best redundancy figures within the analysed range of system.

[25] Similar study was subsequently conducted for an average duration flight of five hours, and it is discussed later in this Chapter (section 4.2.2.3).

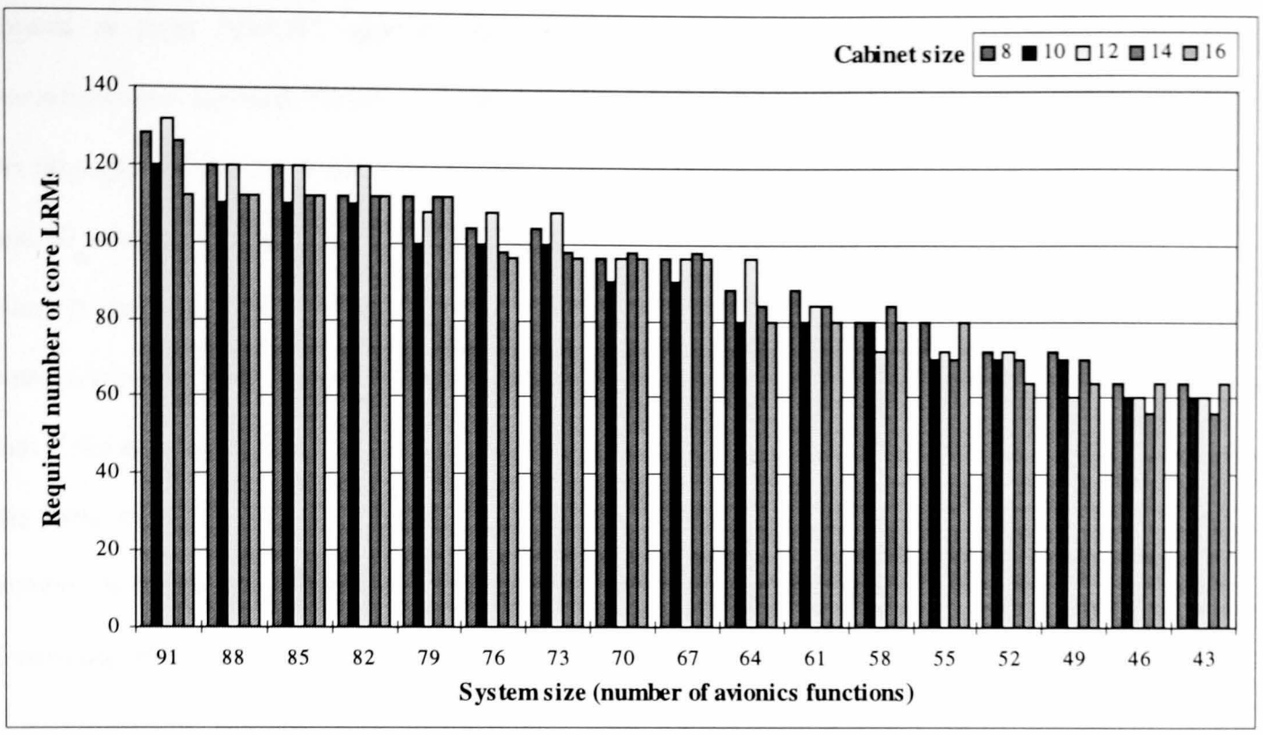



**Figure 4.1. Total number of core LRMs vs. cabinet size vs. system size.**

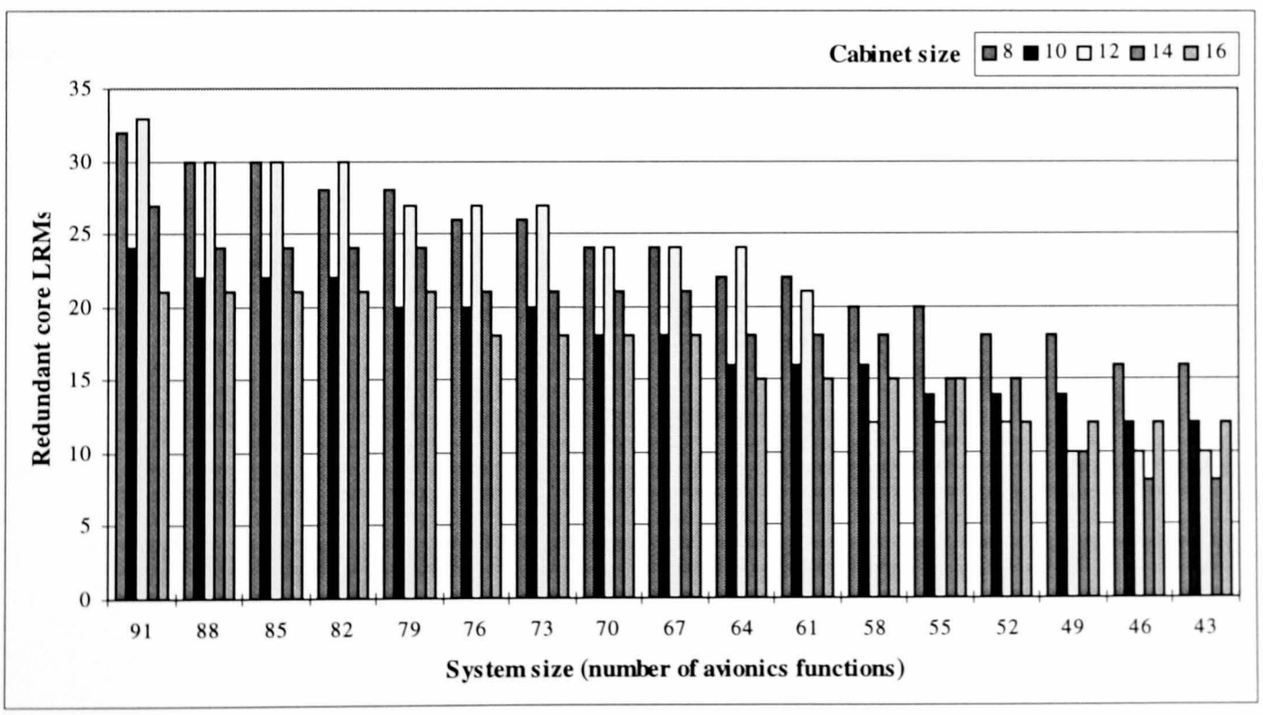



**Figure 4.2. Number of required redundant core LRMs vs. cabinet size vs. system size.**

Generally, the choice of small sizes of eight or fewer core LRMs in the cabinet leads to an increased redundancy overhead. The size of sixteen modules seems to be pretty reasonable for the range of avionics systems being investigated. This agrees with some of the conclusions from the discussion on global and local reconfigurations schemes. However, the reconfiguration of bigger cabinets can be

treated as more "global", and as such it is subject to several drawbacks inherent to the global reconfiguration approach identified in Chapter 1.

As the expected size of avionics systems in the near future can be expected to lie somewhere between 50 and 80 avionics functions, the cabinet size of ten core LRMs (including redundant modules) would be of some preference. If the systems were to grow above about 96 functions (see section 4.2.2 for details) some alternative sizes could become more appropriate, and could be established by a similar study. Note that if the number of functions is lower than 60, the number of twelve core LRMs per cabinet would be the most desirable. In the case of multi-application processing modules, it can be expected that the number of functional groups should not exceed 60 even for systems of hundred or more prime avionics functions, and thus cabinets with twelve core LRMs would be of greatest benefit in such RIMA systems.

As shown in the table below (Table 4.1), the differences between the best and the worst case do not exceed 20% of the total number of processing modules, and for the system sizes of the greatest interest are even lower (less than 15%). That would imply that benefits following a particular cabinet size in the design of a RIMA system, although significant, should not be overemphasised.

| Size<br>S | Best<br>B | B/S<br>(%) | Worst<br>W | W/S<br>(%) | (W-B)/S<br>(%) |
|---|---|---|---|---|---|
| 91 | 21 | 23.08 | 33 | 36.26 | 13.19 |
| 88 | 21 | 23.86 | 30 | 34.09 | 10.23 |
| 85 | 21 | 24.71 | 30 | 35.29 | 10.59 |
| 82 | 21 | 25.61 | 30 | 36.59 | 10.98 |
| 79 | 20 | 25.32 | 28 | 35.44 | 10.13 |
| 76 | 18 | 23.68 | 27 | 35.53 | 11.84 |
| 73 | 18 | 24.66 | 27 | 36.99 | 12.33 |
| 70 | 18 | 25.71 | 24 | 34.29 | 8.57 |
| 67 | 18 | 26.87 | 24 | 35.82 | 8.96 |
| 64 | 15 | 23.44 | 24 | 37.50 | 14.06 |
| 61 | 15 | 24.59 | 22 | 36.07 | 11.48 |
| 58 | 12 | 20.69 | 20 | 34.48 | 13.79 |
| 55 | 12 | 21.82 | 20 | 36.36 | 14.55 |
| 52 | 12 | 23.08 | 18 | 34.62 | 11.54 |
| 49 | 10 | 20.41 | 18 | 36.73 | 16.33 |
| 46 | 8 | 17.39 | 16 | 34.78 | 17.39 |
| 43 | 8 | 18.60 | 16 | 37.21 | 18.60 |

**Table 4.1. Best/Worst cases for core LRM redundancy in RIMA design.**

The discussion so far has been focused strictly on the processing modules without regard to redundancy of power supply modules, gateways and buses. The number of the aforementioned components is related to the number of core LRMs, as well as the number of cabinets (the availability requirements per cabinet increase with the number of cabinets - each of the cabinets must be more available to meet the overall dispatch availability objectives). Thus systems based on smaller cabinets are more likely to require more non-processing modules (power supply, gateways) and buses.

The number of power supply modules is also strongly related to the total number of core LRMs as they are capable of supporting a limited number of modules. Thus a reduction in the total number of (redundant) power supply modules should follow a reduction of the number of core LRMs.

The number of gateways and backplane buses is strictly bound to the number of cabinets. One can expect between two and four gateways and backplane buses to be necessary to meet the safety requirements. As a decrease in the number of cabinets implies lower availability requirements per cabinet, it could lead towards further cost reduction as a lower MTBF of cabinet elements would be required. On the other

hand however, the physical capacity of these devices is strictly limited (e.g. throughput for the data buses) and an increase of the number of processing modules in the cabinet may incite the need for additional buses or gateways. That could again lead to the need further redundancy in order to meet safety objectives.

There are other problems related to big cabinets, e.g. it would be generally more difficult to find a suitable location for their placement on the aircraft. Furthermore, as the number of avionics functions in the system does not in general divide by the number of cabinet active[26] core modules, one can expect some cabinet slots remaining unused. The number of empty slots is likely to be greater for bigger cabinets than for the smaller ones (e.g. if two functions are left over after "filling" all other cabinets, it would imply eleven empty slots in a sixteen core cabinet and only six in a ten core cabinet[27]).

### 4.2.2. Cabinet configuration

In this section cabinet redundancy assessment is conducted in order to establish the required cabinet configuration (ratios of different types of functions, processing modules redundancy, constraints on the criticality of failure combinations, etc.). Results from the previous section are also taken into account when making particular choices.

#### 4.2.2.1. Cabinet size and availability objectives

The assessment of core module redundancy has been conducted for cabinet size of ten core LRMs, what follows the findings shown in Figure 4.1 and 4.2. The choice of ten processing units per cabinet takes into account minimisation of the number of redundant processing modules in the expected range of system sizes between 50 and 80 avionics functions (see Figure 4.2), and avoids possible drawbacks inherent with the management of bigger cabinets. However, the study was also conducted for twelve and sixteen core LRMs cabinets for the purposes of comparison and to allow discussion of some more general tendencies.

---

[26] The number of active processing modules should be understood as the size of the cabinet decreased by the number of redundant (inactive) core LRMs.

[27] Assuming three redundant core LRMs for the sixteen core cabinets, and two for the ten core cabinets.

The configuration of a cabinet has to address the issues of safety objectives and dispatch availability without unscheduled maintenance. To achieve the necessary level of dispatch availability (in this study 99% for the whole system, about 99.9% per cabinet) the system has to tolerate multiple failures, thus the aircraft has to be able to take off with failed modules.

To find all the failure combinations that need to be tolerated, the Markov method with a time interval of 400 hours was used to calculate the probability of a cabinet enduring any of the possible combinations of failures. The probability of the state with no modules failed was the highest in all the studied cases, however it was not sufficient to meet the 99.9% availability objective. In such cases the states with a single and possibly multiple failed modules had to be considered until the required dispatch availability was achieved. That gave the number of failures that have to be tolerated without affecting the safety objectives for any of the avionics functions. The table below (Table 4.2) shows the relation between the cabinet size, tolerated number of failures and the unavailability.

| Tolerance | 10 core LRM (%) | 12 core LRM (%) | 16 core LRM (%) |
|---|---|---|---|
| 4 failed | 0.000071 | 0.000215 | 0.001108 |
| 3 failed | 0.002934 | 0.006699 | 0.023118 |
| 2 failed | 0.083930 | 0.149369 | 0.358357 |
| 1 failed | 1.587466 | 2.268090 | 3.914403 |
| 0 failed | 18.126925 | 21.337214 | 27.385096 |

**Table 4.2. Unavailability with a tolerance for multiple failures.**

To meet the 99% system availability objective, it can be easily shown from the table above (Table 4.2), that if the number of cabinets does not exceed twelve for ten core LRM cabinets ( $13 \times 0.00294 > 1$ , twelve cabinets implements about 96 functions) or six for twelve core LRM ( $7 \times 0.149369 > 1$, six cabinets implement about 60 functions), a cabinet needs to tolerate two failures, i.e. an aircraft needs to be able to take off with up to two failed modules per cabinet and still support all the required avionics functions. A sixteen core LRM cabinet needs to exhibit a tolerance of three failures (up to 43 cabinets, i.e. about 550 avionics functions). With a tolerance of just two failed modules per cabinet, the RIMA system would be able to use not more than two sixteen core LRM cabinets (about 28 functions or functional groups) to meet the 99% system availability objectives.

#### 4.2.2.2. Cabinet configuration requirements for one hour flights

The Markov method was run for a one hour time interval for cabinets with different numbers of faulty modules[28] at the start of the computation. That gave the probabilities of different combinations of failures to occur within a one hour flight providing that a certain number of core LRMs were already not operating at the aircraft take-off. It also constituted a check against the safety requirements and allowed combinations of failures to be found that are not allowed to lead to a loss of particular avionics functions. For example, if a particular combination is not extremely improbable (less than $10^{-9}$ per flight hour) it must not lead to a loss of a critical function and thus to a catastrophic mode of failure[29].

The cabinet required tolerance, i.e. the number of failures that must not lead to a loss of a certain type of avionics function is given in the following tables (Table 4.3, Table 4.4 and Table 4.5). These were established for cabinets consisting of 10, 12 and 16 core LRMs with MTBF equal to 20,000 hours each and for various requirements regarding dispatch availability. Note that all values are absolute, i.e. they do not relate to the number of faulty modules at the start of processing, (for example, if "Major function tolerance" equals three in Table 4.3 it means that three out of ten processing modules failing in a cabinet must not lead to major failure conditions).

| Number of faulty LRMs at the start | Catastrophic function tolerance | Hazardous function tolerance | Major function tolerance | Minor function tolerance | Dispatch unavailability (%) |
|---|---|---|---|---|---|
| 0 | 2 | 2 | 1 | 0 | 18.12692 |
| 1 | 3 | 2 | 2 | 1 | 1.58747 |
| 2 | 4 | 3 | 3 | 2 | 0.08393 |
| 3 | 5 | 4 | 4 | 3 | 0.00293 |
| 4 | 6 | 5 | 5 | 4 | 0.00007 |

**Table 4.3. Tolerance figures for 10 core LRM cabinet.**

[28] The number of faulty processing modules corresponds to the tolerance required to achieve a necessary dispatch availability. After 400 hours an aircraft has to be able to take of with n failed modules.

[29] For minor failure conditions the probability of $10^{-3}$ per flight hour or less was achieved with figures as stated in the tables in section 2.2. Note that there are no formal requirements for failure probability of such conditions.

| Number of faulty LRMs at the start | Catastrophic function tolerance | Hazardous function tolerance | Major function tolerance | Minor function tolerance | Dispatch unavailability( %) |
|---|---|---|---|---|---|
| 0 | 2 | 2 | 1 | 0 | 21.33721 |
| 1 | 3 | 3 | 2 | 1 | 2.26809 |
| 2 | 4 | 4 | 3 | 2 | 0.14937 |
| 3 | 5 | 4 | 4 | 3 | 0.00670 |
| 4 | 6 | 5 | 5 | 4 | 0.00021 |

**Table 4.4. Tolerance figures for 12 core LRM cabinet.**

| Number of faulty LRMs at the start | Catastrophic function tolerance | Hazardous function tolerance | Major function tolerance | Minor function tolerance | Dispatch unavailability (%) |
|---|---|---|---|---|---|
| 0 | 2 | 2 | 1 | 0 | 27.38510 |
| 1 | 3 | 3 | 2 | 1 | 3.91440 |
| 2 | 4 | 4 | 3 | 2 | 0.35836 |
| 3 | 5 | 5 | 4 | 3 | 0.02312 |
| 4 | 6 | 6 | 5 | 4 | 0.00111 |

**Table 4.5. Tolerance figures for 16 core LRM cabinet.**

The tolerance (redundancy) numbers would not need any discussion for non-reconfigurable IMA systems. Simply put, each of the core LRMs would need as many back-up modules as it is required by the dispatch availability and safety requirements (see tables 4.3, 4.4 and 4.5). However, with dynamic in-flight reconfiguration some considerable savings on redundant core modules can be achieved, that are discussed later in this section. The discussion on non-reconfigurable systems and a comparison between non-reconfigurable and reconfigurable systems is given later in sections 4.2.3 and 4.2.4.

The required number of redundant core LRMs is constrained by the necessary tolerance of the minor functions (the lower bound), as in case of a minor function loss neither of the remaining operating processing modules would reconfigure. As in RIMA redundant core LRMs are capable of undertaking any of the cabinet functions, then there is no need for separate spare modules for each of the minor functions. Thus, irrespectively of the number of minor functions, the lower bound on the number of redundant cores will be the same for given cabinet size, safety requirements and dispatch availability.

Moreover, the lower bound of the cabinet core LRM redundancy should not be affected by the number of more critical functions. In the case where a major function module fails, its function can be undertaken

by any of the redundant modules or one of the minor function modules. For example, if minor functions require two redundant modules and major functions require three redundant modules, only two redundant modules and one minor function module will be required to meet the requirements.

In a similar fashion the necessary minimal configuration can be established for all types of functions. If in the example above catastrophic functions require four back-up core LRMs, it would imply the need for at least two minor functions, or one minor and one major function. In the latter case, the following assumption on the combinations of failures would also be required to hold: none of the combinations of failures that involves a loss of one minor and one major function will lead to catastrophic failure conditions. This simple requirement, however, is likely to be true for all avionics systems.

> Commentary:
> Core LRM redundancy requirements can be analysed in an alternative way. Assuming the situation as in the 12 core cabinet table (Table 4.4) for the dispatch unavailability of 0.14937% in 400 hours, it can be deduced that the cabinet will need to tolerate two failures without losing any of the avionics functions (minor functions tolerance). Therefore the cabinet will require at least two redundant processing modules. In the case of a third failure, the system is permitted to lose a minor function, thus a minor function module will reconfigure to provide backup. If there was not a single minor module in the cabinet another redundant core would be required at some additional cost. Therefore an optimal cabinet would need two redundant core modules and at least one minor function. In the case of the fourth failure, the system is permitted to lose either another minor or a major function. That implies a need for at least two minor functions or one minor and one major function in the cabinet. Any subsequent failures are extremely improbable and are allowed to lead to a loss of any of the functions.

The following tables (Table 4.6 and Table 4.7) describe minimal configuration requirements for RIMA cabinets based on the results from tables Table 4.3 and Table 4.4 for "dispatch availability level 2" (an aircraft needs to be able to take off with two failed core LRMs per cabinet).

| *Minimal numbers of modules/functions (alternatives)* | | | | |
|---|---|---|---|---|
| Spare | Minor | Major | Hazardous | Catastrophic |
| 2 | 2 (or more) | any | Any | any |
| 2 | 1 | 1 (or more) | any | any |
| 2 | 1 | 0 | 1 (or more) | any |
| *Combinations of failures constraints (respectively to alternatives above)* | | | | |
| 2 minor failure combinations are not catastrophic. | | | | |
| 1 minor and 1 major failure combinations are not catastrophic. | | | | |
| 1 minor and 1 hazardous failure combinations are not catastrophic. | | | | |

**Table 4.6. Configuration requirements for 10 core LRM cabinets.**

| *Minimal numbers of modules/functions (alternatives)* | | | | |
|---|---|---|---|---|
| Spare | Minor | Major | Hazardous | Catastrophic |
| 2 | 2 (or more) | any | any | any |
| 2 | 1 | 1 (or more) | any | any |
| *Combinations of failures constraints (respectively to alternatives above)* | | | | |
| 2 minor failure combinations are not hazardous or catastrophic. | | | | |
| 1 minor and 1 major failure combinations are not hazardous or catastrophic. | | | | |

**Table 4.7. Configuration requirements for 12 core LRM cabinets.**

By analogy, the following table (Table 4.8) describes minimal configuration requirements for sixteen core LRM cabinets and dispatch availability level three (see Table 4.5). The choice of a higher availability level for this cabinet size follows the availability study described in Table 4.2, as avionics systems based on sixteen core cabinets with dispatch availability level lower than three could not in practice meet the 99% availability objectives for 400 hours operation without maintenance.

| *Minimal numbers of modules/functions (alternatives)* | | | | |
|---|---|---|---|---|
| Spare | Minor | Major | Hazardous | Catastrophic |
| 3 | 2 (or more) | any | any | any |
| 3 | 1 | 1 (or more) | any | any |
| *Combinations of failures constraints (respectively to alternatives above)* | | | | |
| 2 minor failure combinations are not hazardous or catastrophic. | | | | |
| 1 minor and 1 major failure combinations are not hazardous or catastrophic. | | | | |

**Table 4.8. Configuration requirements for 16 core LRM cabinets.**

As the ratios of different types of functions can be expected to be in the order of "1:2:1:1" (minor-major-hazardous-catastrophic) with cabinet sizes as analysed (10, 12, 16), the requirements for the minimal numbers of particular types of functions in the cabinet can easily be satisfied. Various other ratios of functions criticality are considered later in this chapter (section 4.3 and 4.4).

The constraints related to combinations of failures from the tables above (Table 4.6, Table 4.7 and Table 4.8) in their strongest form assume that a combination of a minor and a major failure is not hazardous, and a combination of a major and a hazardous failure is not catastrophic. Although in practice this is almost always true, there may exist cases where such assumptions do not hold (e.g. the loss of all navigation systems is major and the loss of all communication systems is major, but the combination of these two is catastrophic). However, such cases are rare and they can easily be avoided by placing the functions in different cabinets.

Note also that for 10 core LRM cabinets the criticality requirements for various combinations of failures are even weaker, which makes the choice of ten processing modules per cabinet more attractive.

#### 4.2.2.3. Cabinet configuration requirements for five hours flight time

In order to assess the cabinet configuration requirements for an average duration flight of five hours (typical for a long range airliners), similar studies to those in section 4.2.2.2 were conducted for an extended time interval. As explained in section 4.2.2.1, to achieve the required dispatch availability a cabinet needs to provide full functionality with two processing modules failed at the aircraft take off for the cabinet size of ten or twelve core LRMs, and with three core LRMs failed for the size of sixteen. As the dispatch conditions have already been assessed (sections 4.2.2.1 and 4.2.2.2) the analysis described in this section is focused purely on the cabinet configuration requirements, similar to those in tables Table 4.6, Table 4.7 and Table 4.8. The average flight duration was assumed to be five hours (as for Airbus A340), and the Markov analysis was conducted for such a time interval. The results were obtained for core LRMs of 20,000 hours MTBF.

Table 4.9 shows the tolerance requirements for different cabinet sizes and the level two dispatch availability (see Table 4.3 for comparison). Note that for ten core LRMs cabinets the required tolerance

increased only for the hazardous functions when compared with the figures from Table 4.6. For twelve core LRMs cabinets the results are identical with those from Table 4.7 for the availability level two. Also, the figures for the cabinet size sixteen are again identical with those from Table 4.8 for an availability level three.

| Cabinet size | Catastrophic function tolerance | Hazardous function tolerance | Major function tolerance | Minor function tolerance |
|---|---|---|---|---|
| 10 | 4 | 4 | 3 | 2 |
| 12 | 4 | 4 | 3 | 2 |
| 16 | 5 | 5 | 4 | 3 |

**Table 4.9. Tolerance figures for cabinets for an average duration flight.**

The results shown in the tables below (Table 4.10 and Table 4.11) should be understood similarly to the results from the tables above (Table 4.6, Table 4.7 and Table 4.8).

| *Minimal numbers of modules/functions (alternatives)* | | | | |
|---|---|---|---|---|
| Spare | Minor | Major | Hazardous | Catastrophic |
| 2 | 2 (or more) | any | any | Any |
| 2 | 1 | 1 (or more) | any | Any |
| *Combinations of failures constraints (respectively to alternatives above)* | | | | |
| 2 minor failure combinations are not catastrophic or hazardous. | | | | |
| 1 minor and 1 major failure combinations are not catastrophic or hazardous. | | | | |

**Table 4.10. Configuration requirements for 10 and 12 core LRM cabinets (average duration flight).**

The table above (Table 4.10) shows that for an average duration flight of five hours, the configuration requirements for a cabinet consisting of ten core LRMs are only slightly more demanding than for a one hour flight, and they are identical for cabinets containing twelve processing modules. Thus the results of the assessment of cabinet configuration requirements for these sizes from section 4.2.2.2 are also valid for the five hour average duration flight.

| *Minimal numbers of modules/functions (alternatives)* | | | | |
|---|---|---|---|---|
| Spare | Minor | Major | Hazardous | Catastrophic |
| 3 | 2 (or more) | any | any | any |
| 3 | 1 | 1 (or more) | any | any |
| *Combinations of failures constraints (respectively to alternatives above)* | | | | |
| 2 minor failure combinations are not hazardous or catastrophic. | | | | |
| 1 minor and 1 major failure combinations are not hazardous or catastrophic. | | | | |

**Table 4.11. Configuration requirements for 16 core LRM cabinets (average duration flight).**

Since the cabinet configuration requirements from Table 4.11 above are identical with those from Table 4.8, the results from section 4.2.2.2 are also valid for the five hour average duration flight of an aircraft with a RIMA system based on sixteen core LRMs cabinets.

#### 4.2.2.4. Conclusion

From the study above several guidelines towards the choice of RIMA cabinets configuration with respect to their size and the system size can be made:

- very small cabinet sizes (eight or less) are impractical, as the number of redundant core modules per cabinet is relatively high when compared with the cabinet size
- although big cabinets (e.g. sixteen core LRMs) provide the capacity for further growth of avionics systems in terms of being able to operate with the same number of redundant processing modules per cabinet for a wide range of system sizes, they may need more redundant gateways and data buses in order to meet the safety and availability objectives
- it can be expected that big cabinets will be more difficult to install in an aircraft
- in big cabinets many processing modules would be sharing the limited capacity of the ARINC 659 backplane that could lead, in critical cases, to significant delays or data bus throughput exhaustion
- bigger cabinets generally require more redundant processing modules to meet the availability requirements, however their ratio of the number of redundant processing modules to the cabinet size can be better for some system sizes
- constraints on criticality of combinations of failures are weaker for ten core LRM cabinets rather than for twelve or sixteen core LRM cabinets within the range of analysed system sizes
- there is no significant difference in cabinet configuration requirements for a one hour flight and a five hour duration flight for systems employing 20,000 MTBF core LRMs.

### 4.2.3. Traditional systems

In order to perform a fair comparison between Reconfigurable IMA architectures and the traditional systems (whether black box based or IMA), the redundancy of the later systems was assessed in a similar manner as RIMA systems in sections 4.2.1 and 4.2.2.

For different criticality functions the redundancy required to meet the safety objectives has been identified in a similar fashion it has been accomplished for RIMA systems. With MTBF equal to 20,000 hours it is relatively easy to show that catastrophic functions need to be implemented by three processing modules (or black boxes), hazardous and major functions by two modules and minor functions by a single module. These requirements have to be met at the start of a flight, however, if 99% availability in 400 hours was required, there some additional dedicated redundancy will be required.

To assess the additional redundancy enforced by the dispatch availability objectives, the Markov analysis was conducted for various configurations of black boxes for the time interval of 400 hours. The analysis gave the unavailability figures for each of the configurations that can be understood as probability thresholds, i.e. if a function needs to be available with probability $P_k$ it has to be in configuration $C_n$ (n processing modules implementing the function). That is similar to the analysis of RIMA cabinets, where a cabinet has to support all the required functions with some modules failed prior to aircraft take off.

> Commentary:
> If a function is required to be available with a probability of 99.95% in 400 hours, it could imply that its configuration must tolerate say, two failed modules without affecting the reliability figures. If it was a catastrophic function it would have to be implemented by five (3+2) modules. Thus the Markov analysis would have to be performed for the configuration of 5 black boxes. Note that the availability figures relating to the same number of failures depend on the total number of modules in the configuration. This is related to the fact that the Markov analysis gives accurate answers, whilst fault tree analysis gives only an upper bound of the system unavailability [35].

The simplest approach to achieve system availability not worse than $A$ would be to divide the permitted unavailability $U=1-A$ equally between all functions. However, in such a case it is unlikely that $U_f$ (function unavailability) would be just under the required threshold and thus the overall system availability actually achieved could be much greater than required.

Commentary:
If $U_f$ equals, say, 0.025 (system of 40 functions and 99% availability) and the thresholds obtained from a previous reliability and availability study indicate that for $U<0.001$ the configuration requires three additional redundant modules, for $U<0.1$ it requires two additional redundant modules and for $U>0.1$ it requires a single additional redundant module, the necessary redundancy would be three. However, the achieved availability (assuming the same configuration for all forty functions) would be at least 99.96% ($(1-40\times0.001)\cdot100\%$) and thus much greater than the required one (i.e. the number of redundant modules would be higher than the possible minimum).

To perform a fair comparison between RIMA and a traditional system the unavailability factor was distributed unevenly between functions according to the optimisation algorithm devised specifically for this purpose.

The optimising algorithm attempts to move particular functions to higher availability thresholds (thus reducing the number of redundant modules), without exceeding the unavailability limits. The new unavailability of a function will be higher than the old one by the lowest possible factor that leads to a change of the configuration. Such factors are lowest for minor functions and highest for catastrophic ones, which is related to the total number of modules implementing the function. The algorithm finishes when the unavailability limit is exhausted or none of the configurations can be changed without breaching the safety requirements (minimal configurations).

Table 4.12 shows the minimal redundancy requirements for traditional systems of the same sizes as the RIMA systems analysed in sections 4.2.1 and 4.2.2, with the assumed ratio of different function types: 1:1:2:1[30]. Note that the table below shows only the number of redundant modules i.e. the total number of modules minus the number of avionics functions (the relation of one function to one processing module, although difficult to realise in practice[31], is highly desirable).

[30] Minor : major : hazardous : catastrophic.

[31] For one-to-one relation between avionics functions and processing modules the module's MTBF would have to be in the order of $10^9$ hours to meet the safety requirements for most critical functions, or modules would have to be able to perform multiple functions at the same time.

| Unavailability | | | | | | |
|---|---|---|---|---|---|---|
| System size | 1% | 2% | 4% | 6% | 8% | 10% |
| 43 | 115 | 106 | 90 | 84 | 83 | 82 |
| 46 | 125 | 116 | 99 | 91 | 90 | 89 |
| 49 | 131 | 122 | 104 | 95 | 94 | 93 |
| 52 | 141 | 132 | 114 | 104 | 102 | 101 |
| 55 | 153 | 144 | 126 | 114 | 110 | 109 |
| 58 | 159 | 150 | 132 | 119 | 114 | 113 |
| 61 | 169 | 160 | 142 | 129 | 121 | 120 |
| 64 | 175 | 166 | 148 | 134 | 125 | 124 |
| 67 | 186 | 176 | 158 | 143 | 133 | 132 |
| 70 | 197 | 188 | 170 | 153 | 144 | 140 |
| 73 | 205 | 194 | 176 | 158 | 149 | 144 |
| 76 | 215 | 204 | 186 | 169 | 158 | 151 |
| 79 | 222 | 210 | 192 | 174 | 163 | 155 |
| 82 | 233 | 220 | 202 | 185 | 172 | 162 |
| 85 | 245 | 232 | 214 | 197 | 183 | 173 |
| 88 | 252 | 238 | 220 | 202 | 188 | 178 |
| 91 | 263 | 248 | 230 | 213 | 197 | 187 |

**Table 4.12. Minimal redundancy of traditional systems.**

If there were no requirements for the system availability, the total number of redundant modules could be calculated from the following equation

$$\text{eq. 1} \qquad N_{redundant} = 3 \times N_{catastrophic} + 2 \times (N_{hazardous} + N_{major}) + N_{minor} - N_{functions}$$

Where $N_{criticality}$ denotes the number of functions of given *criticality*. The table below (Table 4.13) shows the minimal possible configurations of traditional avionics systems with no availability requirements for black box MTBF of 20,000 hours and the same ratio of functions criticality as before.

| $N_{functions}$ | 43 | 46 | 49 | 52 | 55 | 58 | 61 | 64 | 67 | 70 | 73 | 76 | 79 | 82 | 85 | 88 | 91 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N_{total}$ | 84 | 91 | 95 | 102 | 110 | 114 | 121 | 125 | 132 | 140 | 144 | 151 | 155 | 162 | 170 | 174 | 181 |
| $N_{redundant}$ | 41 | 45 | 46 | 50 | 55 | 56 | 60 | 61 | 65 | 70 | 71 | 75 | 76 | 80 | 85 | 86 | 90 |

**Table 4.13. Minimal configurations for traditional avionics systems.**

### 4.2.4. Comparison

In this section a comparison of RIMA systems and non-reconfigurable avionics systems with respect to the number of redundant processing modules is performed in order to assess the possible benefits of

system capacity for dynamic in-flight reconfiguration. Note that only the processing modules redundancy is taken into account, and similar analysis would be required to assess these factors for other types of modules.

The figure below (Figure 4.3) shows the required numbers of redundant core LRMs (or black boxes for traditional systems) for different system sizes and various availability requirements. The data values of the series captioned as "average RIMA "C" (99%)" has been calculated as an average number of required redundant modules across the range of analysed cabinet sizes (8, 10, 12, 14 and 16) for RIMA architecture "C" (see Figure 4.2 and Table 4.1), and as such is sub-optimal. This has been intended to favour the traditional systems in order to avoid any bias towards the preference of RIMA. Note also, that the percentage rates stated in series names relate to system availability in 400 hours, and the minimal IMA/BB system (Integrated Modular Avionics/Black Boxes non-reconfigurable design) data has been calculated for no availability objectives.

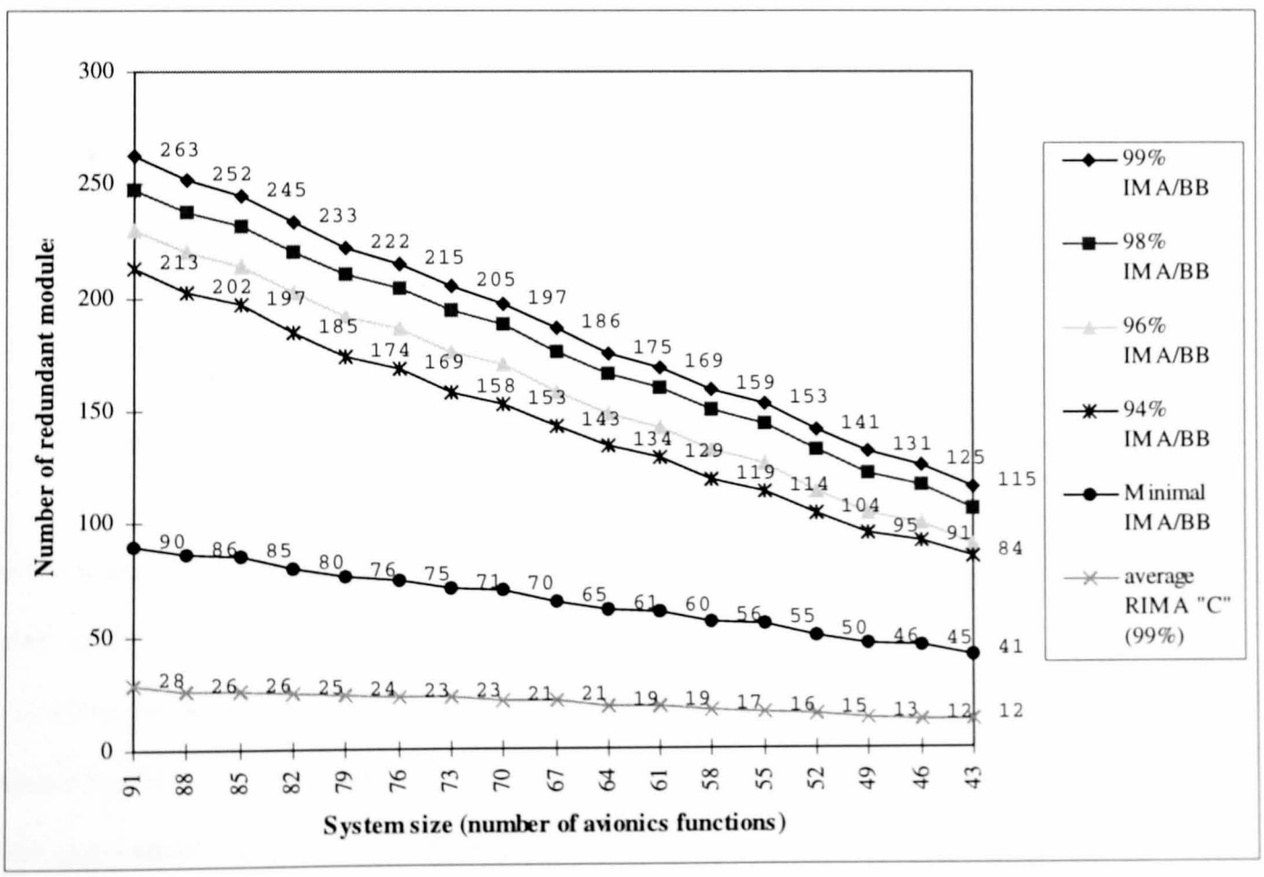



**Figure 4.3. Processing module redundancy comparison for RIMA and traditional avionics systems.**

As can be seen in the graph above, the requirements for redundant core LRMs (processing modules) are lowest for RIMA systems, even when compared with the optimised minimal IMA/BB system. It becomes

clear that traditional systems are less attractive as they require almost ten-fold more core LRM redundancy to meet the same objectives as the reconfigurable systems.

The following Figure 4.4 shows the increase of the number of redundant modules relative to the "average RIMA system". Note that in the best case a traditional system with no availability requirements (in practice, it would be virtually certain that such a system will require some unscheduled maintenance) employs about three times as many redundant processing modules as RIMA avionics systems with 99% availability.

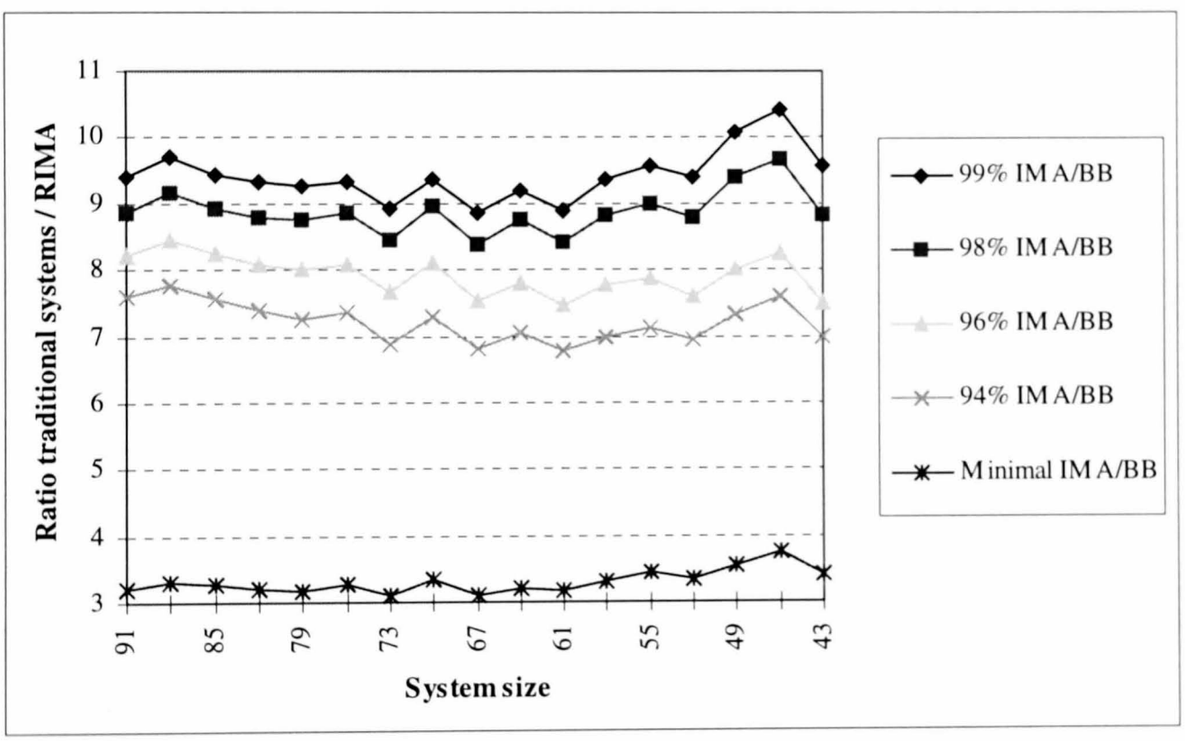



**Figure 4.4. Relative increase in processing module redundancy (traditional systems / RIMA).**

In the process of comparison between the traditional and reconfigurable systems an explicit effort has been made not to overestimate the traditional systems redundancy and not to underestimate the redundancy of RIMA avionics systems. Thus, all the results discussed above were obtained with the intention of favouring the traditional systems (i.e. all functions that were left after calculating 1:1:2:1 ratio were assumed to be minor, i.e. least redundant).

The above figures Figure 4.3 and Figure 4.4 have been based on the previous numerical analysis (sections 4.2.1, 4.2.2 and 4.2.3), and they show that Reconfigurable Integrated Modular Avionics can lead to major savings in terms of the reduced number of processing modules required to implement an avionics system,

as well as in terms of improved system availability. In order to assess the overall benefits other system components such as gateways and data buses would also need to be analysed. However, based on the analysis of core LRM redundancy, RIMA systems can be expected to perform significantly better than avionics systems not employing dynamic in-flight reconfiguration.

### 4.2.5. Systems with between-flights reconfiguration only

In the previous sections the reduction of the number of redundant processing modules following the capacity for dynamic in-flight reconfiguration was discussed, and a comparison between RIMA and non-reconfigurable systems was made. In this section some consideration is given to a particular type of IMA that employs between-flights reconfiguration techniques.

Reconfiguration between-flights allows for changes of the system/cabinet module-function assignments on the ground only. Thus in the case of any failure during the flight the system must respond with some dedicated redundancy. It is clear that such systems must employ at least the same number of redundant core LRMs as the minimal configuration of non-reconfigurable avionics systems discussed in section 4.2.3 and Table 4.10 in order to meet the safety requirements for the flight.

Moreover, the availability figures for such systems would not change if between-flight reconfiguration was to be introduced. In the minimal configuration a loss of a single module leads to a violation of safety requirements. No on-ground reconfiguration (change of the module-function assignment) would bring the reliability to the required level, as there would not be enough dedicated redundancy in the system.

Based on the above arguments a claim can be made that systems with between-flight reconfiguration cannot achieve lower processing module redundancy than the minimal configuration of non-reconfigurable systems, and that in such configuration their availability figures would not improve respectively to the non-reconfigurable minimal system.

The benefits of the between-flights reconfiguration occur only if the considered system is not minimal. In such a case the capacity for on-ground changes of the system module-function assignments can greatly contribute towards system availability figures, which can be expected to be higher than in the traditional

non-reconfigurable systems. The avionics systems with on-ground reconfiguration can be considered as an intermediate step between non-reconfigurable IMA systems and RIMA with dynamic in-flight reconfiguration, but they do not fully exploit the benefits of reconfiguration.

## 4.3. RIMA architecture "C"

The assessment presented in the previous sections was generically valid for many possible RIMA designs. In this section, however, particular issues related to the RIMA architecture "C" are discussed.

### 4.3.1. Software replication

In order to be able to reconfigure the system in the case of a module failure the appropriate application software must be available either within the system or from some external source. As discussed before, there are many ways to provide the software store, however, when the RIMA design "C" is being considered the idea of processing modules storing copies of different avionics functions in their non-volatile memory appear the most practical.

In such a solution the memory requirements and the choice between global and local reconfiguration approaches is extremely crucial. A simple analysis shows that for architecture "C" using the core LRMs to store the software, the global reconfiguration approach is unfeasible in many ways:

- technologically - too high requirements for memory to be presently implemented in the limited volume of a processing module
- economically - even if future electronics allow such implementation it would be very costly
- with respect to the system safety - the system reconfiguration process becomes very complex.

When local reconfiguration schemes are being considered the issue of software replication within the cabinet has to be addressed. Several questions relate to this problem:

- How many copies should exist for each function?
- Which modules should store which applications?
- What are the memory requirements?

The figures shown in Table 4.9 give the answer to the first question for an average duration flight. In order to be able to tolerate four failures for a certain function criticality type, for each function of such criticality at least four copies have to be stored in the cabinet. Thus the total number of software copies within a cabinet can be easily calculated as:

$$\text{eq. 2} \qquad N_{copies} = r_1 \times N_1 + r_2 \times N_2 + r_3 \times N_3 + r_4 \times N_4$$

where $r_x$ describes replication requirements for the x-th function criticality, and $N_x$ represents the number of such functions. The following tables (Table 4.14, Table 4.15 and Table 4.16) show the number of required software copies in relation to the cabinet size and the ratio of different function criticalities within the cabinet. Note that the following tables assume the tolerance figures as shown in Table 4.9, and the ratios of different functions are chosen to reflect the most probable options (although some of the options are intentionally rather unrealistic, see option 'e' in Table 4.14, option 'e' in Table 4.15 and options (d) and (e) in Table 4.16).

| Option | Catastrophic | Hazardous | Major | Minor | $N_{copies}$ |
|---|---|---|---|---|---|
| (a) | 2 | 3 | 2 | 1 | 28 |
| (b) | 2 | 3 | 1 | 2 | 27 |
| (c) | 3 | 3 | 1 | 1 | 29 |
| (d) | 2 | 4 | 1 | 1 | 29 |
| (e) | 1 | 5 | 1 | 1 | 29 |

**Table 4.14. Number of software copies for a 10 core LRM cabinet.**

| Option | Catastrophic | Hazardous | Major | Minor | $N_{copies}$ |
|---|---|---|---|---|---|
| (a) | 2 | 3 | 3 | 2 | 33 |
| (b) | 2 | 4 | 2 | 2 | 34 |
| (c) | 3 | 4 | 2 | 1 | 36 |
| (d) | 3 | 4 | 1 | 2 | 35 |
| (e) | 3 | 5 | 1 | 1 | 37 |

**Table 4.15. Number of software copies for a 12 core LRM cabinet.**

| Option | Catastrophic | Hazardous | Major | Minor | Needed |
|---|---|---|---|---|---|
| (a) | 3 | 5 | 3 | 2 | 58 |
| (b) | 4 | 5 | 2 | 2 | 59 |
| (c) | 3 | 4 | 3 | 3 | 56 |
| (d) | 5 | 6 | 1 | 1 | 62 |
| (e) | 4 | 7 | 1 | 1 | 62 |

**Table 4.16. Number of software copies for a 16 core LRM cabinet.**

The number of required software copies strongly influences the requirements for the core LRM memory capacity. Before these requirements can be properly assessed, a choice has to be made regarding the way of supporting the software store functionality by processing modules, and the question of which modules should store the applications also needs to be answered. However, the answer to this question is not as straightforward as to the previous one.

Generally two approaches can be distinguished. The first allows modules to exchange software between each other (implying the use of some software bus), the second avoids the necessity of a software bus but does not allow software migration.

In the first solution presented in greater detail in the next section, every active module can store the same number of software copies and the redundant modules can store yet another copy (they are not required to perform any avionics function at the system start-up). Thus, even the catastrophic function modules can provide copies of software for other core LRMs (clearly a catastrophic function module cannot be requested to perform a less critical function as that would lead to the loss of an aircraft, it can only provide required software for other modules). This approach allows a very good utilisation of cabinet memory (each core LRM is assumed to have the same memory capacity), exhibiting, however, some drawbacks related to the requirement for a software downloading bus.

In the second approach (discussed in section 4.3.3) each module can only store software for functions of lower criticality than the one being performed by the module. As there are no means for software migration, critical function modules would not store any software, hazardous modules would store only copies of critical functions, etc. Clearly the redundant processing modules can store software for functions of any criticality. In this solution the system suffers from non-optimal memory utilisation,

avoiding at the same time potential problems related to the software downloading bus. However, in the case of a module recovery from a transient fault, it could be beneficial if critical modules stored software for some less critical functions. In such a situation a recovered critical module could take over some previously lost less critical function (clearly its original, critical, function would be performed by another module at that time).

The memory requirements depend on the chosen approach, as well as the cabinet size and the ratio of different avionics functions criticality in the system being considered.

### 4.3.2. RIMA cabinets with a dedicated Software Bus

Architecture "C" based RIMA systems with cabinets employing a software downloading bus lead generally to lower requirements for non-volatile memory per processing unit. On the other hand, however, the need for software fetching from a remote module can introduce some delays in reconfiguration and it would require a dedicated software downloading bus.

At present there is no standard in the avionics industry for a software bus that could be used for the purposes of dynamic in-flight software downloading. The ARINC 629 and ARINC 659 seem to be too slow for such an operation, and thus a new type of a data bus would probably have to be developed. That would of course be costly and it would decrease the attractiveness of this approach. Also, avionics systems with a newly introduced software downloading bus would have to be additionally certified for their safety.

Moreover, the impact of the additional reconfiguration delays would have to be assessed. Clearly the length of the reconfiguration delay depends on the speed of the software bus, so a very fast software bus could help in the elimination of this problem. Furthermore, provided that some redundant software copies exist within the cabinet, it may be possible to reconfigure the cabinet in the event of a processing module failure and then to allow software migration .

Low memory requirements and good processing module memory utilisation are not the only benefits of this approach. The reconfiguration scheme can be kept simple and based on the "hot" and "cold" standby

modes as suggested in [31]. Also, since each module can store any applications there are no problems with the feasibility of the software-module assignment. Generally, however, the time and cost related drawbacks of a software downloading bus based approach suggest that other ways of supporting the software replication should be sought and employed.

The table below (Table 4.17) shows the relation between the number of supported software copies not including the software for non-redundant modules primary applications, and the processing module available non-volatile memory for different cabinet sizes.

| | 4 MB per core LRM | 5 MB per core LRM | 6 MB per core LRM |
|---|---|---|---|
| 10 core LRM cabinet | up to 32 copies | up to 42 copies | up to 52 copies |
| 12 core LRM cabinet | up to 38 copies | up to 50 copies | up to 62 copies |
| 16 core LRM cabinet | up to 51 copies | up to 67 copies | up to 83 copies |

**Table 4.17. Number of supported software copies vs. available core LRM memory.**

Note that the values in the table above (Table 4.17) are based on the assumption of two redundant core LRMs for the cabinet sizes of ten and twelve processing modules, and on three redundant core LRMs for the cabinet size of sixteen.

### 4.3.3. RIMA cabinets without a dedicated Software Bus

With the elimination of the software downloading bus the reconfiguration scheme can be imposed on top of an existing IMA system without any changes to the original design[32], which would possibly avoid the need for extensive certification of the underlying hardware. Furthermore, fetching the application software from local memory should prove to be significantly faster than with the use of a data bus, and thus it should lead to potentially much faster reconfiguration. Also, the scheme should become less complex as no additional code related to handling of the data bus based software downloading will be required. However, the solution exhibits a major problem related to the processing module memory requirements.

---

[32] Only the elimination of dedicated I/O modules could be seen as an alteration.

As already mentioned, the whole inactive[33] memory of a catastrophic function modules is left unused in this approach. Also, the hazardous modules can store at most as many software copies as there are catastrophic functions (only a single copy of each function can be stored by a particular core LRM, and clearly not more copies than the memory capacity). Thus in this case the memory requirements depend not only on the cabinet size but also on the ratio of avionics functions of different criticality within the cabinet. Moreover, even if the cabinet effective memory capacity is big enough to store the required number of copies the feasibility of the assignment of software copies to processing modules has to be verified.

The following tables (Table 4.18, Table 4.19 and Table 4.20) show the software storage capacity figures for different sizes of cabinets and for the ratios of different types of avionics functions as indicated in Table 4.14, Table 4.15 and Table 4.16. They also show the memory requirements for a core LRM in order to make the assignment feasible (see Appendix A for examples of assignments for particular ratios of avionics functions). Note that the values shown with the strike-through type are lower than the number of software copies required for the corresponding ratio of different avionics functions (see Table 4.14, Table 4.15 and Table 4.16 for definitions of each option).

| Option | 4 MB / module | 5 MB / module | 6 MB / module | 7 MB / module | 8 MB / module | Feasible for |
|---|---|---|---|---|---|---|
| (a) | ~~23~~ | 28 | 33 | 38 | 43 | 5 MB |
| (b) | ~~23~~ | 28 | 33 | 38 | 43 | 5 MB |
| (c) | ~~23~~ | ~~27~~ | 31 | 35 | 39 | 6 MB |
| (d) | ~~22~~ | ~~26~~ | 30 | 34 | 38 | 6 MB |
| (e) | ~~18~~ | ~~22~~ | ~~26~~ | 30 | 34 | 7 MB |

**Table 4.18. Processing module memory requirements for 10 core cabinets.**

[33] Non-volatile memory not used for storing and state updating of the currently performed function.

| Option | 4 MB / module | 5 MB / module | 6 MB / module | 7 MB / module | 8 MB / module | feasible for |
|---|---|---|---|---|---|---|
| (a) | ~~29~~ | 36 | 43 | 50 | 57 | 5 MB |
| (b) | ~~28~~ | 34 | 40 | 46 | 52 | 5 MB |
| (c) | ~~29~~ | ~~34~~ | 39 | 44 | 49 | 6 MB |
| (d) | ~~29~~ | ~~34~~ | 39 | 44 | 49 | 6 MB |
| (e) | ~~26~~ | ~~30~~ | ~~34~~ | 38 | 42 | 7 MB |

**Table 4.19. Processing module memory requirements for 12 core cabinets.**

| Option | 4 MB / module | 5 MB / module | 6 MB / module | 7 MB / module | 8 MB / module | feasible for |
|---|---|---|---|---|---|---|
| (a) | ~~42~~ | ~~50~~ | 58 | 66 | 74 | 6 MB |
| (b) | ~~39~~ | ~~51~~ | 58 | 65 | 72 | 7 MB |
| (c) | ~~42~~ | ~~51~~ | 60 | 69 | 78 | 6 MB |
| (d) | ~~33~~ | ~~43~~ | ~~53~~ | ~~58~~ | 63 | 8 MB |
| (e) | ~~33~~ | ~~43~~ | ~~48~~ | ~~53~~ | ~~58~~ | 9 MB |

**Table 4.20. Processing module memory requirements for 16 core cabinets.**

Note that for the tables above (Table 4.18, Table 4.19 and Table 4.20) the memory requirements were calculated to meet the safety and availability objectives as discussed before (see Table 4.9 for the replication figures for various criticality functions).

Note also, that in the formal requirements [3], [36] for avionics systems there are no safety objectives for minor modes of failure. In this research a threshold of the failure probability of $10^{-3}$ per flight hour has been chosen as the safety objective for such minor functions. Should a higher probability be allowed the memory requirements could be reduced (in the most extreme case - no copies of minor functions - the reduction could be significant). The minor functions replication would, however, affect the overall system availability figures (clearly, if no back-up copies for minor functions were stored each failure of a minor function module would lead to some system degradation).

As indicated before, some of the options stated in the three tables above are highly unrealistic, and as such could be ignored. In this case 6 MB of non-volatile memory would be required from each processing module to implement RIMA systems based on 10 or 12 core LRM cabinets. For 16 core LRM cabinets 7 MB of memory per processing module should suffice.

To reduce the total memory requirements for a cabinet, the on-board memory installed into catastrophic function modules could be restricted to a size required to perform a single function (unless the capacity for module recovery was to be considered, see section 4.3.1). However, that would implicitly lead to some dedication between modules and avionics functions, and thus the cost of providing and storing different types of processing modules could actually be higher than the cost of a few megabytes of additional memory.

In this approach a single core LRM may be required to store copies of many functions of the same criticality, for example a single hazardous function module may store the software for multiple catastrophic functions. This situation does not violate the safety requirements as a loss of any of the core LRMs leads to a loss of at most one copy of an avionics function. Thus N failures of processing modules will occur before losing an avionics function, where N is the function replication number. As the replication numbers are chosen to comply with the safety requirements (see Table 4.9), the assignment of many backup copies of the same criticality to a single processing module should be considered safe.

It has to be stated that the solutions not based on a software downloading bus offer some attractive features (fast and simple reconfiguration, validity of certification obtained for ARINC 653 architecture "C" based IMA systems) at the cost of additional on-board non-volatile memory. Thus, this approach seems to be more promising for implementation of RIMA systems rather than the approach based on a software downloading bus as described in section 4.3.2.

### 4.3.4. Conclusions

Two major approaches to implementation of RIMA systems based on the ARINC 651 architecture "C" have been discussed. It was concluded that the solution avoiding the use of a software downloading bus (section 4.3.3) is more promising than the one employing a bus for software fetching. The cost of development of a software downloading bus, and the cost of the certification process in the later solution would be much higher than the cost of additional non-volatile memory in the first instance. Moreover, the reconfiguration scheme is likely to be more simple and much faster in designs not employing software downloading buses and not requiring software migration.

All considerations regarding reconfiguration of architecture "C" based systems assumed the use of the local reconfiguration approach, as the global approach has already been identified in as practically difficult or even unfeasible. It appears that systems avoiding the use of software downloading buses for dynamic in-flight reconfiguration are presently more desirable. However, if memory requirements are to grow significantly in the future (for example due to the introduction of much more sophisticated avionics functions) such that their growth could not be balanced by a similar growth in electronics (memory capacity), the systems employing solutions similar to that discussed in section 4.3.2 could be preferred.

## 4.4. RIMA architecture "D"

The main difference between the RIMA architecture "C" and RIMA architecture "D" is the existence of Application Modules (AMs) in the latter approach. In RIMA an application module provides software storage for avionics functions (software replication), as well as the functionality to allow updates and storage of the functions state as required (in architecture "C" such functionality is provided by the processing modules). Clearly, a RIMA architecture "D" design needs to employ some software downloading bus in order to allow dynamic in-flight reconfiguration, and thus it is exposed to problems similar to those identified in section 4.3.2.

### 4.4.1. Availability and reliability

The availability and reliability analysis from section 4.2 of this chapter is generally valid for architecture "D" based RIMA systems. However, some additional issues related to the introduction of application modules into the cabinet design have to be addressed.

As the processing modules used to implement RIMA architecture "D" are not required to perform the software store functionality their design is expected to be more simple than in the architecture "C" and thus potentially more reliable. The application modules would also be much simpler than core LRMs in architecture "C", and thus they can also be expected to be more reliable. Generally all modules used to implement architecture "D" based RIMA systems are likely to be more reliable and thus the availability and reliability figures should be better than in the case of architecture "C" based RIMA systems with the same number of LRMs per cabinet.

On the other hand, however, the number of line replaceable modules per cabinet will be higher for architecture "D" based systems. Clearly, if in the architecture "C" there were 10 core LRMs per cabinet the equivalent architecture "D" system would employ 10 + N modules, where N represents the number of application modules.

The actual reliability and availability study was conducted for 10 + 2 and 10 + 3 configurations of architecture "D" based RIMA systems, and the MTBF of core and application modules varied from 20,000 hours to 30,000 hours. In some cases it was assumed that both types of modules have the same MTBF parameters, however, the more probable situation where application modules have higher MTBF was also assessed. The following table (Table 4.21) shows the availability figures for different cabinet configurations and varying module MTBF parameters obtained with the use of the previously mentioned Markov method for the maintenance free time interval again of 400 hours.

The availability objectives taken into account whilst calculating the figures in Table 4.21 are explained below:

- core LRMs - similarly as in section 4.2 for 10 core LRM cabinets - at least 8 processing modules have to be working at the end of the 400 hours period of time
- application modules - at least two AMs have to be working at the end of the 400 hours period of time.

| MTBF in $10^3$ hours | | | | | | Cabinet availability | |
|---|---|---|---|---|---|---|---|
| Processing modules | | | Application modules | | | Configuration | |
| 20 | 25 | 30 | 20 | 25 | 30 | 10+2 | 10+3 |
| ✓ | | | ✓ | | | 95.9983% | 99.8001% |
| ✓ | | | | ✓ | | 96.7694% | 99.8414% |
| ✓ | | | | | ✓ | 97.2869% | 99.8640% |
| | ✓ | | | ✓ | | 96.8079% | 99.8811% |
| | ✓ | | | | ✓ | 97.3256% | 99.9037% |
| | | ✓ | | | ✓ | 97.3433% | 99.9218% |

**Table 4.21. Availability figures for 10+2 and 10+3 configurations in architecture "D".**

The figures from the table above (Table 4.21) show clearly that it is impossible for the system to reach the 99% availability objective with the 10+2 configuration. However, in the 10+3 configuration the system

can employ multiple cabinets and still meet the 99% dispatch availability requirements. The actual number of cabinets varies from 5 (for the least available option), through 6, 7, 8 and 10 (for the four following options) and finally reaches 12 for the option employing processing and application modules with MTBF equal to 30,000 hours.

### 4.4.2. Software replication

Because of a different way of supporting the software store, problems related to software replication for RIMA systems based on the architecture "D" should be approached in a different manner than for the architecture "C" based systems.

In the case of RIMA systems employing application modules to implement the software store, a failure of a core LRM does not affect the number of software copies stored within a cabinet. In order to meet the safety requirements it has to be shown that it is extremely improbable to lose a catastrophic function processing module after losing all application modules (no reconfiguration is possible in such a case), and to show similar compliance with safety requirements [3] for other less critical functions.

The Markov method was used for configurations shown in Table 4.21 assuming that two of the core LRMs were already faulty at the aircraft take-off, and that only two of the application modules were operating. The time interval was set to five hours in order to obtain the reliability figures for an average duration flight for a long range aircraft. The results show that for either of the configurations and regardless of the chosen MTBF parameters (within the range from 20,000 to 30,000 hours) the chance of losing any function (i.e. losing a processing module after losing all AMs) is extremely improbable. Thus it is the dispatch availability objective that formulates the strongest requirements towards the system.

Moreover, the probability figures related to a loss of particular combinations of core LRMs during a five hour average duration flight are only slightly higher than for architecture "C" cabinets employing core

LRMs of the same MTBF (20,000 hours). The difference is insignificant and does not introduce the need for any changes into the cabinet configuration as discussed in section 4.2.2.[34]

### 4.4.3. Conclusions

The study discussed in section 4.4 shows that architecture "D" based RIMA systems are exposed to the same problems with software downloading buses as architecture "C" based solutions presented in section 4.3.2. However, the memory requirements in the latter design are expected to be higher than in the case of the architecture "D" based solution.

As each avionics function in a ten core LRM cabinet based on the architecture "D" design would be replicated only three times (corresponding to three application modules), each of the AMs would require about 8 MB[35] of non-volatile memory (24 MB total). The total cabinet memory requirements would also have to account for approximately 1 MB of non-volatile memory per each of the cabinet core LRMs (10 MB total), thus they add up to some 34 MB of non-volatile memory per cabinet.

In the architecture "C" based cabinets the memory requirements depend on the ratio of different types of avionics functions, however, it can be expected that each of the processing modules would require not less than 4 MB of non-volatile memory. That would imply at least 40 MB of memory per cabinet in the architecture "C" based designs employing a software downloading bus.

Thus, if a software downloading bus is required, the RIMA architecture "D" design appears to be the preferred choice. Moreover, reconfiguration schemes employed into this design are likely to be more simple than in the case of the architecture "C" and a software downloading bus based solutions. Also, the software running on processing modules can be less complex as there is no need for a core LRM to provide copies of software to another modules. The application modules also provide a convenient and

---

[34] The difference was usually in the order of 0.01% of the relevant probability figure and did not lead to the violation of safety requirements, for example the difference in probabilities is of the order of $10^{-12}$ per flight hour for catastrophic functions.

accessible store for function state parameters and the functionality for function state updating (both tasks are performed by core LRMs in architecture "C").

## 4.5. Systems redundancy conclusions

In the previous sections of this chapter different designs of reconfigurable avionics systems were discussed with respect to their availability, reliability and configuration. The notion of system redundancy was simplified to the redundancy of processing modules. In an optimal avionics system the number of processing modules should be equal to the number of avionics functions or functional groups, such that no two modules would be required to perform the same task. In these terms each module duplicating or backing-up a task has to be considered redundant[36].

The number of processing modules in RIMA architecture "D" is identical to that of RIMA architecture "C". The total number of modules employed by both designs and the number of redundant modules differ only by the number of AMs in the architecture "D" based systems. Since the redundancy of traditional non-reconfigurable systems was much higher than the redundancy of the RIMA architecture "C" based systems (see section 4.2.4), the redundancy of architecture "D" based systems is expected to lie somewhere between the two previously assessed designs (see the figures below – Figure 4.5 and Figure 4.6).

---

[35] Assuming again that the size of each application is of the order of 1 MB.

[36] In the case of RIMA architecture "D", application modules are essential for system reconfiguration, and as such they have to be treated as redundant modules (even though they do not provide the processing power).

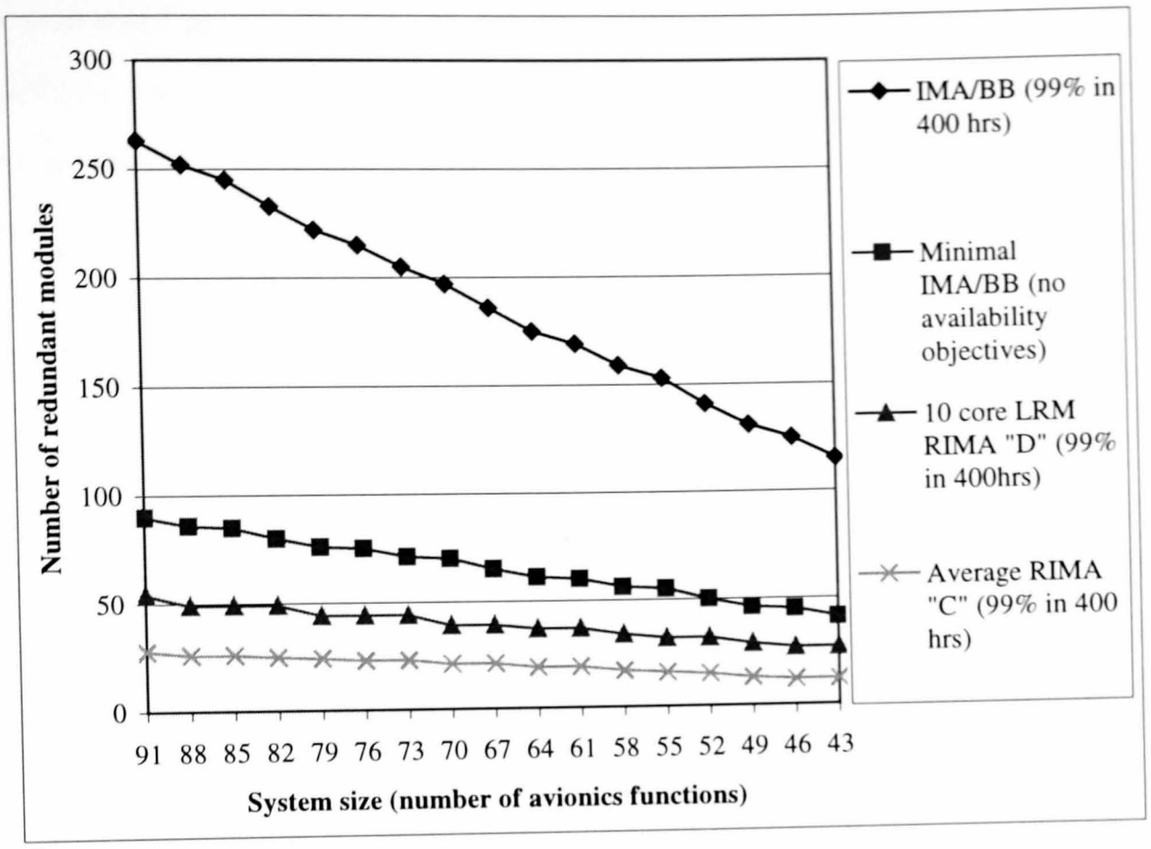



**Figure 4.5. Redundancy figures of RIMA "C" and "D" architectures vs. non-reconfigurable systems.**

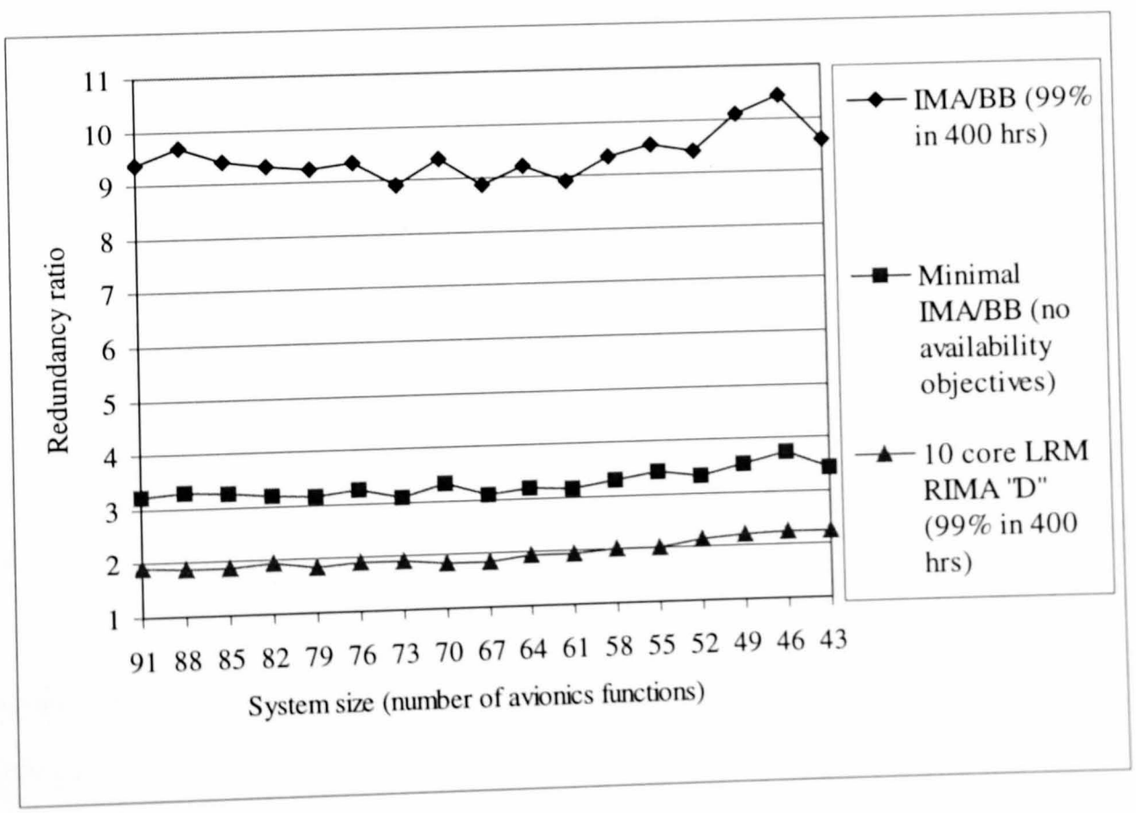



**Figure 4.6. Increase in processing module redundancy relative to an average RIMA "C".**

The figures above (Figure 4.5 and Figure 4.6) show clearly that although the architecture "D" based RIMA systems have higher redundancy than architecture "C" based Reconfigurable IMA, their

redundancy is still by far better than the traditional non-reconfigurable avionics systems. Also, the availability of RIMA "D" systems is aimed to be about 99% in 400 hours, while similar requirements addressed towards non-reconfigurable avionics systems make their redundancy figures worse by almost an order of magnitude.

The redundancy analysis was focused on the level of processing modules, with some additional considerations on application modules. It could be argued that since IMA and RIMA systems employ other non-processing components such as gateways or backplane buses the comparison is unfavourable for the traditional non-reconfigurable black box avionics systems. However, a black box itself constitutes not only a processing module but also contains an interface to the global data bus (a gateway module provides the interface for the whole cabinet, i.e. about 2 gateways for several processing modules), and its own power supply (again in (R)IMA cabinets one power supply module supports multiple core LRMs). Furthermore, in the case of excessive communications between some avionics functions the capacity of the ARINC 629 data bus in traditional black box systems could be exhausted and some additional data buses would have to be provided, thus the existence of the backplane buses could be balanced in a more detailed comparison.

It has to be concluded, that although the analysis conducted in this chapter was restricted to the processing module redundancy level, it has shown a general tendency that the redundancy in RIMA systems would be significantly lower than in non-reconfigurable avionics systems and that their availability is generally higher. In the case of some systems (for example 99% available RIMA and 99% available non-reconfigurable avionics systems) the redundancy figures differ by approximately an order of magnitude. Moreover, the minimal non-reconfigurable system discussed in section 4.2.3 employing about three times as many redundant processing modules as RIMA architecture "C", is likely to require maintenance several times within the 400 hour period of time, while a corresponding RIMA system still has a 99% chance of requiring no maintenance in this time.

A more detailed analysis of the cost of ownership would be required to assess accurately the advantages and disadvantages of different designs[37]. However, it is believed that due to the previously explained relations, the results would exhibit tendencies similar to those presented in this chapter, and would thus strongly favour the choice of the Reconfigurable Integrated Modular Avionics. If no software downloading bus is required, the overheads (cost) associated with reconfiguration should be low and the benefits of reduced redundancy and improved availability should result in significant cost savings.

[37] As date relating to the actual cost of avionics modules is considered sensitive and confidential by the industry, despite arranging a number of meetings with various companies the author was unable to gather sufficient information to perform the cost of ownership analysis.

A more detailed analysis of the cost of ownership would be required to assess accurately the advantages and disadvantages of different designs[37]. However, it is believed that due to the previously explained relations, the results would exhibit tendencies similar to those presented in this chapter, and would thus strongly favour the choice of the Reconfigurable Integrated Modular Avionics. If no software downloading bus is required, the overheads (cost) associated with reconfiguration should be low and the benefits of reduced redundancy and improved availability should result in significant cost savings.

[37] As date relating to the actual cost of avionics modules is considered sensitive and confidential by the industry, despite arranging a number of meetings with various companies the author was unable to gather sufficient information to perform the cost of ownership analysis.

Moreover, as such devices would have an impact on the whole cabinet, any design errors, hardware and software failures, or data corruption could endanger the integrity of the whole system.

Finally, due to its complexity and required reliability the controlling device and its software can be expected to be very expensive, and thus it would reduce the redundancy and cost benefits related to the introduction of Reconfigurable Integrated Modular Avionics systems discussed in Chapter 4.

The above arguments indicate the autonomous reconfiguration methods as being possibly more reliable and more cost beneficial than non-autonomous schemes. Furthermore, since the global reconfiguration approach has shown to be highly impractical, all considerations in this chapter are valid for locally operating autonomous reconfiguration schemes.

### 5.2.1. Autonomous reconfiguration

In order to eliminate single points of failure, the reconfiguration scheme must not rely on any single controlling module. Several issues have to be addressed when autonomous reconfiguration schemes are being considered:

- inter-module communications, i.e. intra-cabinet exchange of information avoiding undesirable interference between processing modules (see section 5.2.1.1)
- consistency and synchronisation of reconfiguration processes running on different core LRMs, i.e. the effect of a possible loss of phase-synchronisation due to a module (or modules) not detecting an event or detecting a non-existent one (see section 5.2.1.2)
- independence and equality of processing modules, i.e. no processing module can overrule decisions of other modules, although each module can make its own decision based on the information from other modules (see section 5.2.1.3)
- conditions for activation of the reconfiguration process (see section 5.2.1.4)
- invalid activation of the reconfiguration process, e.g. a module detects a non-existent failure (see section 5.2.1.5)
- maintenance of reconfiguration data consistency (see section 5.2.1.6).

#### 5.2.1.1. Inter-module communication

Particular reconfiguration schemes can be designed avoiding any need for inter-module communications, where the failure or recovery events can be detected based on the monitoring of the activity on the cabinet data busses. For example, all core LRMs could monitor the data bus for transactions related to particular avionics functions in order to determine the current state of the cabinet. Depending on the actual protocol of the data bus used, various techniques could employed to implement this task, that could be based on monitoring of the transmission windows and the data freshness flag for ARINC 659 protocol, or alternatively the required information could be included in the message labels proposed in [10].

On the other hand, some reconfiguration schemes may require explicit communication channels for purposes such as event detection (see section 5.2.4.3 and 5.2.5.3) or maintenance of reconfiguration data consistency[38] (see section 5.2.1.6), that would be exchanged within the cabinet over the common communication media such as the backplane bus.

Different communication levels related to inter-module information exchange are widely discussed in section 2.3.5 of [10]. In the case of RIMA architecture "C" with a software downloading bus (SDB) or in the case of RIMA architecture "D", the software downloading bus could also be used for message exchange. However, since reconfiguration is essential for the correct operation of the cabinet in case of failures, the SDB would have to be considered safety-critical in a similar manner to the backplane bus.

#### 5.2.1.2. Synchronisation of reconfiguration processes

Although every effort has to be made in order to obtain reliable event detection, the issue of possible loss of synchronisation of reconfiguration processes running on separate processing modules has to be considered. This could be related, for example, to invalid failure or recovery detection performed by one of the core LRMs.

> Commentary:
> Although reconfiguration processes operate asynchronously on separate processing modules, their schemes, however, need to be phase-synchronised (i.e. they all have to be in appropriate pre and post-detection phases). Consider a simple

[38] Although implementation of procedures for maintenance of data consistency leads to somewhat increased software complexity, it is expected to improve the robustness of a reconfiguration scheme (see sections 5.2.1.6 and 5.2.7).

> reconfiguration scheme with multiple backup modules for critical functions. In the situation where the first backup module (core A) does not notice that the function is lost, it will not attempt to reconfigure, and the second backup module (core B) will be expected to assume the role of the first backup[39]. Being the immediate backup for the lost function the module will finally reconfigure, however, should in the meantime core A detect the loss of the function it will also attempt to perform the same application. The failure and recovery events constitute the clock for reconfiguration scheme phase-synchronisation.

It is possible that in the event as in the commentary above where two modules perform identical functions due to the loss of phase-synchronisation, some adjudication techniques could be employed in order to change the avionics function performed by one of the modules. Such procedures should be separated from the main reconfiguration algorithm in order to avoid unnecessary software complexity and integration, and could exploit smart actuators to indicate such modes of failure. Although theoretically the robustness of the reconfiguration algorithm could benefit from the employment of some method of adjudication, it would practically lead to an introduction of a new class of possible modes of failure (a module erroneously notices that some other core LRM performs the same function, and stops performing it). Also, since the modes of failure as discussed in the above commentary would generally lead only to a loss of non-critical functions (unless an extremely improbable number of failures occurred in a cabinet), the benefits following implementation of an adjudication method into a reconfiguration scheme should not be overestimated.

Reconfiguration schemes should be designed to withstand the hazard of single message upsets related to faulty event detection due to a loss or misunderstanding of a single message (see sections 5.2.4.3 and 5.2.5.3). The reconfiguration algorithm should allow autonomous phase re-synchronisation of processing schemes based either on explicit communication between processing modules or on some non-communication based methods, providing that the cause of loss of synchronisation is not permanent.

In the case of re-synchronisation of processes via communication channels every module could, for example, broadcast a message confirming successful event detection, upon reception of which each of the core LRMs could make its own decision based on majority voting or similar techniques. Explicit synchronisation via inter-module communication channels is likely to significantly increase the traffic on the backplane data bus. In order to avoid a phase-synchronisation related backplane bus traffic overhead,

---

[39] The immediate (first) backup core LRM will be assumed faulty.

some other autonomous re-synchronisation methods can be incorporated into the algorithm, that would eliminate the possibility of single message upsets.

Commentary:
A simple approach to implicit phase re-synchronisation can be implemented with event detection based on multiple messages. Figure 5.1 shows the situation where the first backup module (core *A*) fails to immediately detect the loss of a function (does not notice the loss of the first message). The next backup module (core *B*) does not experience the same problem, and thus reconfiguration schemes on those two modules lose phase-synchronisation. However, as the event detection algorithm operates on multiple messages (in this example the loss of four consecutive function messages has to be noticed to detect the loss of an avionics function), both schemes re-synchronise autonomously without inter-module communication after a period of time.

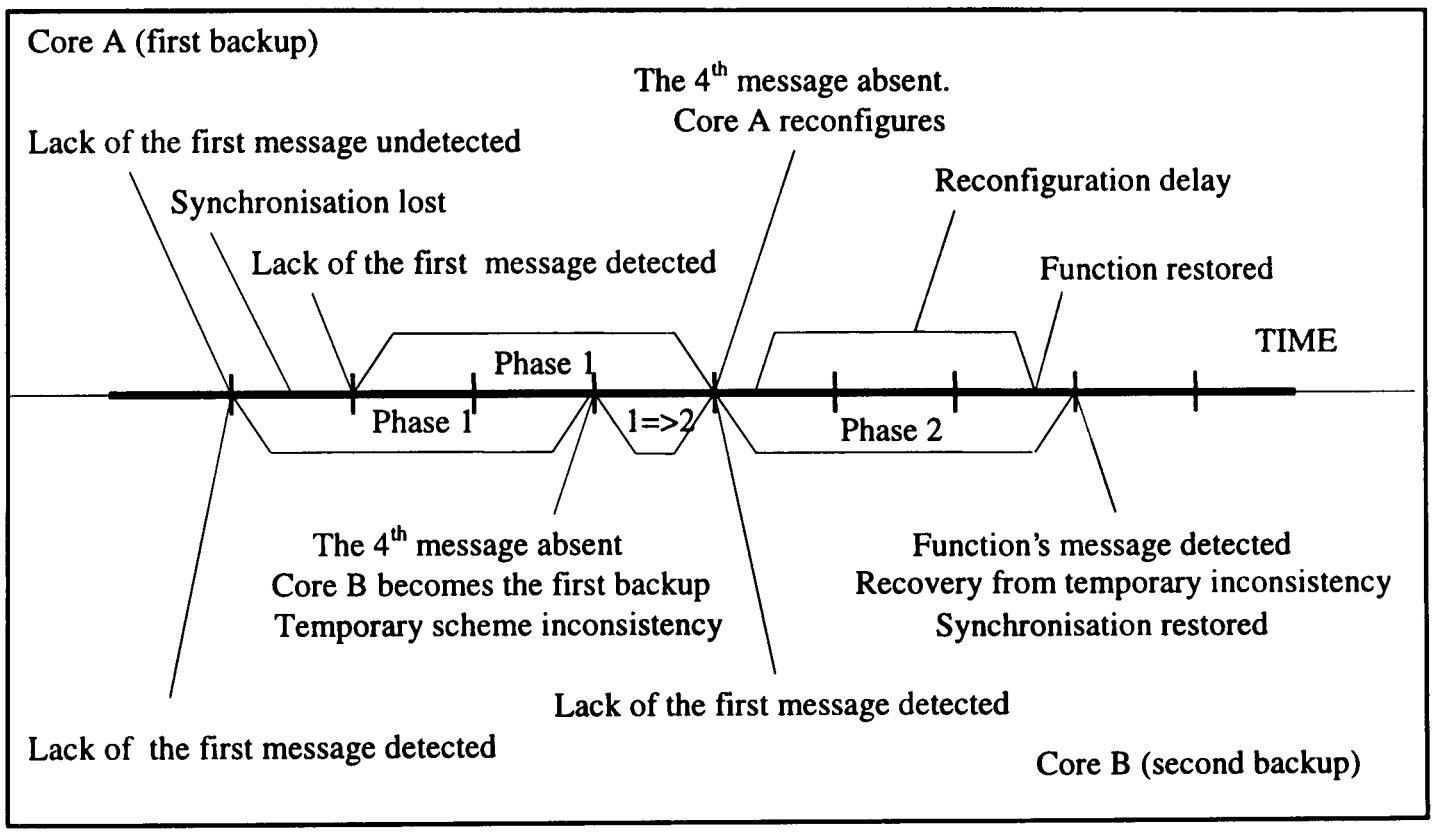



**Figure 5.1. Example of implicit phase re-synchronisation.**

In principle non-communication based re-synchronisation methods would depend on backplane bus monitoring.

### 5.2.1.3. Independence and equality

As the reconfiguration process needs to be autonomous it must not rely on any single controlling device. Moreover, all processing modules participating in the reconfiguration scheme need to be independent and of identical priority in order to eliminate the possibility of a faulty module overriding the decisions of a

working unit. It is also undesirable to allow multiple modules to override decisions of any particular core LRM, even if the decision follows majority voting.

> Commentary:
> Algorithms exploiting the possibility of majority decision override are expected to suffer a significant penalty in increased software complexity, which in turn could lead to errors in the reconfiguration software. In the situation where the reconfiguration algorithm exhibits the powerful property of decision overriding, any design or implementation errors could potentially lead to the malfunction of the whole cabinet.

In the case of independent and equal processing modules it is the algorithm synchronisation procedures (section 5.2.1.2) and the reconfiguration data maintenance policy (section 5.2.1.6) that ensure the proper operation of the reconfiguration processes running on separate core LRMs.

#### 5.2.1.4. Conditions for activation of the reconfiguration process

In order to minimise the possibility of invalid activation of the reconfiguration process, the precise conditions that should lead to algorithm activation have to be identified. Although the reconfiguration process is started separately for each processing module, its activation conditions must account for factors related both to the state of the cabinet as a whole, and to the state of the particular module. Reconfiguration of a core LRM can thus be initiated if:

- the module being considered has detected a loss of an avionics function (or a functional group), that was previously performed by another core LRM
- the criticality of the lost function is higher than the criticality of the function performed by the processing module being considered
- the reconfiguration strategy (see section 5.2.1.6) indicates that the core LRM being considered should reconfigure.

The first two conditions are applicable to all reconfiguration schemes and they guarantee that no reconfiguration process will start if no failure was detected, and that a loss of a non-critical function will not lead to reconfiguration of a critical function module. Moreover, the first activation condition states also that a module must not attempt to reconfigure in the case of its own failure. The third condition is strictly specific to the reconfiguration algorithm and it is subject to additional proof of correctness.

#### 5.2.1.5. Invalid activation of the reconfiguration process

The notion of invalid activation of the reconfiguration process is related to the possibility of invalid event detection (see sections 5.2.4.3 and 5.2.5.3 on failure and recovery detection) and subsequently to inconsistencies in the data crucial for the reconfiguration scheme. Although every effort has to be made in order to provide reliable means for failure and recovery detection in RIMA systems, the reconfiguration algorithm itself has to exhibit fail-safe properties in the case of a mis-detected failure or recovery and must ensure safe system behaviour.

The activation conditions, as defined in the previous section, must still be satisfied in order to start the reconfiguration process, for example, a detection of a non-existent failure of some non-critical module must not cause reconfiguration of a more critical core LRM. The possibility of occurrence of the reconfiguration data corruption or data inconsistency leading to invalid compliance with the third activation condition, follows the possibility of various hardware or software problems. In such a case a processing module could undertake wrong actions based on the corrupted data.

Although particular reconfiguration algorithms could allow different levels of data consistency maintenance (see section 5.2.1.6), since these are also subject to malfunction, the reconfiguration algorithm itself should enforce the following fail-safe properties:

- invalid event detection by a particular core LRM must not cause a malfunction of event detection procedures on other processing modules
- reconfiguration data corruption must remain contained within the affected core LRM, and it is not allowed to propagate through the system
- the probability of invalid reconfiguration activation and the consequences of such an event do not violate safety requirements as defined in [37], [3] and [36].

#### 5.2.1.6. Maintenance of reconfiguration data consistency

The concept of reconfiguration data relates to all data structures that may be used by the reconfiguration process in order to determine the module behaviour in the event of another module failure or recovery. Non-auxiliary reconfiguration data, that is specific to the reconfiguration scheme and not its implementation (e.g. the order of reconfiguration), constitutes the notion of the reconfiguration strategy

data[40]. Auxiliary reconfiguration data (e.g. used for event detection - see sections 5.2.4.3 and 5.2.5.3, or avionics function selection on module recovery - see sections 5.2.5.2, 5.2.5.3 and 5.2.5.4) is regarded as reconfiguration scheme non-specific, and implementation dependent.

Note, that the domain of reconfiguration data is included in the notion of the reconfiguration process software which also embraces the executable code. This section focuses on corruption of the reconfiguration data only, as general issues related to avionics systems software are widely addressed in literature (see [37], [3], [36], [10], [2]).

Data structures, internal to the reconfiguration process, are exposed to potential software/hardware failures or some external hazards[41] that may lead to data corruption or data inconsistency[42].

> Commentary:
> Inconsistencies between reconfiguration data used by separate reconfiguration processes may lead to undesirable behaviour of some modules. For example, if a reconfiguration scheme utilises a lookup table in order to determine its actions in the event of a function loss, a corruption of such a strategy table can violate some of the reconfiguration principles. In the example shown in Figure 5.2, the core module performs the most critical function and as such is not intended to be in backup for any other function ("DO NOTHING" entries in the lookup table). Corruption of one of the entries may force the module to reconfigure in case of some non-critical function loss.

The algorithm should provide means not only for fail-safe operation in the case of reconfiguration data corruption or inconsistency (see section 5.2.1.5), but it could also support some data consistency maintenance policy.

One could argue that critical function modules need not run the reconfiguration software, as they are not intended to reconfigure in the event of any failure. Such an approach could avoid certain problems related to reconfiguration data corruption, however, in the event of a transient failure and following

---

[40] Reconfiguration strategy data will be referred to as reconfiguration strategy or, including auxiliary data, as reconfiguration data.

[41] Recent research indicates the phenomenon of neutrino bombardment as a possible source of unpredictable changes of electronic memory locations. Moreover, the probability of occurrence of such an event is estimated to be in the order of $10^{-3}$ per flight hour.

recovery (see sections 5.2.4 and 5.2.5), the module would require to undergo some reconfiguration and recovery procedures in order to restore an avionics function. Therefore, even the critical function modules would require the reconfiguration software to enhance system fault tolerance and its availability. Other implications following a possible introduction of a "special" status for critical function modules are discussed in section 5.2.7.1.

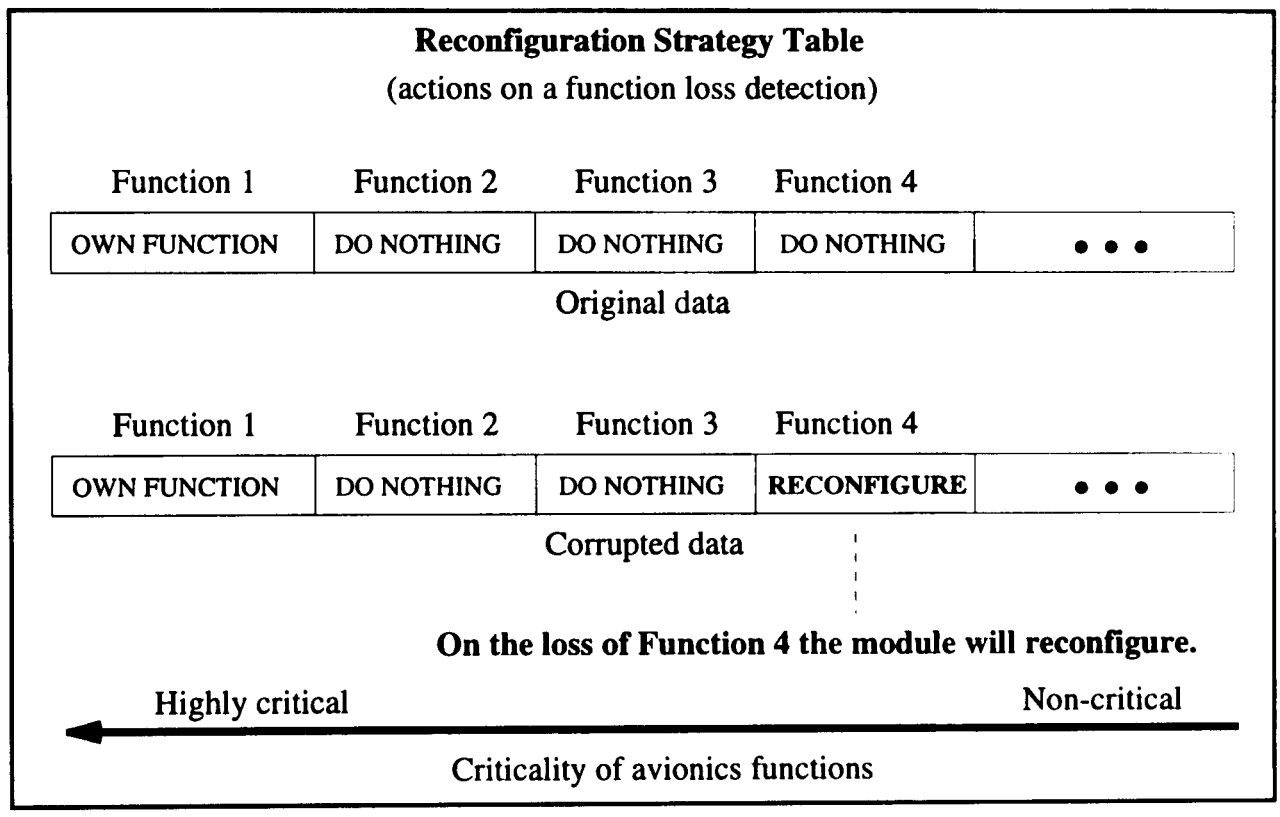



**Figure 5.2. Example of reconfiguration data corruption.**

As modules are independent from each other data coherency can be maintained via inter-module communication channels. For example, the reconfiguration data could be exchanged between modules and majority voted. However, it is essential that each processing module makes its own decision to accept, discard or update the data based on the messages received from other core LRMs, and that none of the modules holds an executive voice in such a process.

Alternatively, each of the processing modules could also store multiple copies of the reconfiguration data in order to avoid problems related to external sources of data corruption[43]. Clearly, it is impossible with this method to avoid internally sourced data corruption, and thus this approach is applicable to

---

[42] Some issues related to data and algorithm corruption and their sources are further discussed in section 5.2.7 of this paper.

[43] See section 5.2.7.1 for a discussion on possible sources of data corruption.

reconfiguration algorithms based on static reconfiguration data (data structures which are not updated during the cabinets operation).

Different reconfiguration schemes may temporarily allow data inconsistency (related to some transitory lack of phase-synchronisation between modules), provided that the reconfiguration algorithm is capable of regaining phase-synchronisation and coherency without inflicting any safety risk, or the possibility of invalid reconfiguration in such circumstances and its consequences meet the JAR/FAR safety requirements.

Some increase of the backplane data bus traffic can be expected due to data consistency maintenance. In order to reduce such a communication overhead, processing modules could verify their data based on some simple consistency checks (e.g. parity checks or data checksums) prior to full data exchange on the backplane bus. The robustness and reliability of methods based on checksum verification is comparable with that of methods based on explicit full data exchange, as current methods for message digest and checksum generation (e.g. MD5) ensure with extremely high probability that no two different messages will have identical signatures [38].

### 5.2.2. Dynamic reconfiguration

As discussed in Chapter 4, only dynamic in-flight reconfiguration can lead to major savings in processing modules redundancy, as opposed to between-flights reconfiguration. However, allowing a module to change its function dynamically during the system operation introduces various problems:

- any reconfiguration delays would in practice imply a loss of an avionics function for some (possibly very short) time, and could lead to an increased crew workload or they could expose an aircraft to some safety hazard (e.g. a loss of the flight control system for a few seconds could lead to the loss of the aircraft)
- long reconfiguration chains following single fault events, i.e. a failure of a catastrophic function module should cause reconfiguration of some non-critical module (redundant or minor), rather than reconfiguration of a hazardous or a major function module, that would in turn cause further reconfiguration.

#### 5.2.2.1. Reconfiguration delays

Reconfiguration delays can be attributed to three main phases of reconfiguration:

- failure detection - before detecting a loss of a particular avionics function, data related to this function must be absent on the data bus for a period of time
- software downloading – the delay contribution of this phase depends on the system architecture and the reconfiguration scheme
- context switching - restoring execution of an application on a new module.

> Commentary:
> It is expected that delays associated to the first phase will be longer in the case of algorithms based on software downloading. In the case of detection of a function loss, a module standing in an immediate backup starts downloading the software in order to restore performance of that function. If the failure detection algorithm does not take into account delays related to this phase, another module will assume that the immediate backup has failed and will attempt to reconfigure to the same function. Furthermore, as the delay required for software downloading is likely to be longer than that required for actual failure detection, algorithms without the software downloading phase are potentially capable of operating with significantly shorter delays.

In order to minimise the reconfiguration delays, a module could imitate performance of the function before completing reconfiguration, for example by putting pseudo-data messages on the backplane bus. Although such a policy could lead to shorter delays related to the failure detection phase, it is undesirable for a module to acknowledge the restoration of performance of an avionics function before downloading the software. In the case of some data bus connection problems or a failure of the application modules, a core LRM might not be able to download the software and consequently to switch the processing environment. At the same time, due to the premature confirmation of successful reconfiguration, other processing modules would not be able to detect such a mode of failure. It is expected, that having the appropriate software downloaded a module will be able to perform it, as the operation of context switching is relatively simple, and it should be provided by the operating system or application executive (APEX) [10].

The delay analysis for a particular reconfiguration scheme must identify the worst possible case for reconfiguration[44] of each function, and such worst case delays must be proven to be shorter than the allowed function absence time.

#### 5.2.2.2. Reconfiguration chains

In the best case a single function loss should lead to reconfiguration of at most one module, i.e. the least critical module should take over the lost function. The reconfiguration scheme should also ensure that a loss of the first two[45] critical functions (catastrophic or hazardous) will lead to reconfiguration of minor or redundant modules.

The reconfiguration schemes developed during this research were designed to operate on the principle of single reconfiguration following a single event of failure, thus reducing the length of the reconfiguration chain to one. In theory this can be always achieved, although for some systems not employing software downloading buses, the non-volatile memory requirements may prove it unfeasible, as the redundant modules will be required to internally store the software for all functions in the cabinet. The strict relationship between the size of the cabinet and the required non-volatile memory indicates again that smaller sizes (see Chapter 4) should be preferred again.

In the case of reconfiguration algorithms with the software downloading phase, it is always possible to ensure that only the least critical processing module will reconfigure in the event of a core LRM failure, as each processing unit is able to download any required software.

### 5.2.3. Determinism and integrity

In order to minimise the risk of malfunction related to high system integration in RIMA, the properties of determinism and integrity have to be requested from the reconfiguration scheme.

---

[44] The worst case should take into account safety requirements, as in the case of multiple failures some algorithms may suffer long reconfiguration delays. These are irrelevant if the probability of such an event is sufficiently low.

[45] The third failure is extremely improbable for cabinets of 10 core LRMs (see Chapter 4) and system capacity for withstanding it exceeds safety requirements.

The property of determinism ensures that the assignment of avionics functions to core LRMs in a cabinet does not depend on the time at which the events of failure and recovery occur. The program executed by a module depends purely on the number, the sequence and the kind of events encountered during cabinet operation. In order to prove the determinism property of a reconfiguration scheme, it has to be shown that the same stable state of a cabinet will be reached if the same sequence of processing modules failures and recoveries occurs regardless of the time intervals between the events.

> Commentary:
> The time intervals between consecutive failures need to be longer than reconfiguration delays in order to eliminate simultaneous or near simultaneous failures. Since reconfiguration delays are likely to be of the order of hundreds of milliseconds all failure and recovery events can be expected to be non-simultaneous (except common cause failures). If two failures are not common cause related, the probability of them occurring within a very short period of time is extremely small, thus any consequences of such event are acceptable within the safety requirements.

As explained above simultaneous failures not related to a common cause[46] are expected to be extremely improbable, and they do not have to be dealt with by the reconfiguration process. Moreover, algorithms designed to deterministically withstand simultaneous failures are likely to be highly complex and will in general lead to longer reconfiguration chains. They may also be unfeasible for certain sets of avionics functions implemented in a cabinet.

> Commentary:
> Even with a synchronous data bus it is impossible to ensure the order of detection of simultaneous failures, as it depends strictly on the time of occurrence. To make the algorithm behaviour deterministic none of the processing modules must be allowed to be in an immediate backup for more than one function (this is not desirable, as the redundant modules should be able to reconfigure first to as many as possible functions). In such solutions some more critical core LRMs would be required to be an immediate backup for some avionics function, and thus long reconfiguration chains would be likely to follow.

It is impossible to withstand simultaneous failures by a class of reconfiguration algorithms with dynamic reconfiguration data updating and dynamically varying backup levels.

---

[46] It should be shown for a RIMA system that a common cause failure of a number of processing units within a cabinet is extremely improbable or the consequences of such event are not catastrophic to the scheme or to the system.

Commentary:
In such algorithms, backup levels for avionics functions are initially assigned to core LRMs, and are changed as failures occur. Backup levels determine whether a module should reconfigure in the case of a function loss (immediate backup) or just update the data (e.g. change from level two to level one). Therefore, if a core LRM was in the backup level one (immediate) for function *F1* and in the backup level two for function *F2*, after the first loss of function *F2* it will be in the immediate backup for both functions. Thus in the case of a consecutive simultaneous loss of functions *F1* and *F2* it will not be able to behave deterministically.

It is desirable to design reconfiguration algorithms whose determinism property is stronger than that defined previously in this section, and in which the assignment of avionics functions to processing modules does not depend on the sequence of encountered failures[47]. Although this property may be very desirable, it is unlikely that such algorithms will be optimal in terms of fast reconfiguration and short reconfiguration chains. Also, their reconfiguration data structures are likely to be very complex and difficult to generate.

The property of reconfiguration algorithm integrity determines, that for every processing module in a cabinet in the event of a function loss the reconfiguration will take place, provided that the activation conditions (see section 5.2.1.4) are satisfied for that module. The property of reconfiguration integrity ensures also, that only the modules for which the reconfiguration activation conditions are satisfied will reconfigure, and no other modules will attempt to do so.

### 5.2.4. Failure

The notion of a processing module failure is of great importance in reconfigurable avionics systems. Therefore, events that are to be classified as failures have to be defined and methods for reliable detection of such events need to be developed.

All considerations in this section are related to the notion of failure as observed in RIMA systems with respect to the reconfiguration process. Moreover, only failures associated with malfunction of processing modules are of interest here, as failures of backplane buses or gateway modules are dealt on the system level and not by the reconfiguration scheme.

---

[47] Such defined determinism will be referred to as "extended" determinism in distinction to an earlier defined "normal" determinism.

Commentary:
In IMA systems the loss of all of the backplane buses or all of the gateways inevitably leads to the loss of the whole cabinet. Such a catastrophic situation is also generically present in the RIMA designs. Although the loss of backplane buses renders the reconfiguration impossible, it also leads to the loss of all of the cabinets functions, which should be considered as more critical. Thus the introduction of the capacity for dynamic in-flight reconfiguration into avionics systems does not influence the safety concerns where backplane bus or gateway module failures are in question.

In the case of a power supply failure, however, the system experiences a simultaneous loss of multiple processing modules, and such an event is capable of triggering the reconfiguration process. Although, a reconfiguration scheme could theoretically be designed to withstand such severe failure conditions, multiple simultaneous failures would in general lead to the algorithm becoming non-deterministic (see section 5.2.3), and as such should be dealt with separately. For instance, multiple redundant power supply modules could be employed in order to bring the probability of a power supply loss to the extremely improbable level. It is expected that the problem of a power supply loss will be dealt with by some hardware replication policy rather than by the reconfiguration software, as the scheme non-determinism seems to be inevitable in such an event.

Other aspects of the notion of failure in highly integrated avionics systems are discussed in [37] and other certification and safety related papers.

#### 5.2.4.1. Definition

With respect to reconfiguration, a failure can be defined as a malfunction of a core LRM that leads to the loss of an avionics function performed by this processing module or to invalid function results. Thus the notion of failure cannot be strictly identified with the notion of a function loss, i.e. a function may still be performed by the module but its results may be meaningless to the system.

Based on the above definition, three different types of processing module failures can be distinguished:

- hard failure - a permanent hardware failure of a processing module that leads to its elimination from further processing
- soft failure - a transient hardware failure or a software failure leading to production of detectable invalid results or to a temporary loss of processing capacity

- latent failure - a software or hardware failure that remains undetected and leads to invalid function results.

It is believed that soft or latent failures are more common in avionics systems than the hard failures, particularly early in the service life of the aircraft.

#### 5.2.4.2. Principles

In non-reconfigurable avionics systems hard and soft failures are dealt with in a similar manner. A faulty module is switched off line permanently in the case of a hard failure, or until it successfully passes self-testing procedures in the case of a soft failure. Although a module that suffers from a soft failure can be brought back to operation, there is a significant probability that allowing this module to perform the same function will result again in a similar failure (especially in the case of corrupted software). The latent failures, as they remain undetected, are not explicitly dealt with, however, it is possible that in the case of redundant systems the invalid results will be overruled by majority voting.

In RIMA, the hard failures can be dealt with in an identical manner to non-reconfigurable avionics systems, where the faulty module is permanently switched off line. In the case of soft failures, however, more robust system behaviour can be achieved with the use of reconfiguration.

A soft malfunction of a core LRM is more likely to be related to data corruption or other software problems than to a hardware problem. As every core LRM in the cabinet is capable of performing any of the cabinet functions, problems related to corrupted function software can be dealt with in a relatively simple way. In such a case, a backup module would undertake the temporarily lost function based on an unaffected copy of the function software fetched either from its own non-volatile memory or downloaded via a software downloading bus[48]. If the soft failed module subsequently passes its self-test procedures and recovers from the failure, it will either reconfigure to perform another function (again the software is likely to be unaffected), or it will operate as a redundant module.

---

[48] Some mechanisms should be implemented into the software downloading process to avoid fetching a corrupted copy of the required software.

There exists a possibility that in the case of a soft failure of a non-critical module, the module will recover to the same function and thus the situation will not be different from the one in non-reconfigurable systems. In this case, however, a repeatable core LRM malfunction could only affect the least critical functions in the cabinet.

If the soft failure was attributed to some hardware problem that may occur again even when unaffected software is used by the module, due to reconfiguration the "weak" module is likely to be requested to perform some non-critical function. It is also possible that the hardware failure will affect some functions but not others.

#### 5.2.4.3. Detection

A failure detection method is expected to be based on backplane data bus monitoring for the lack of handshake messages from a module, the lack of data related to a function or for a combination of the two. It is undesirable, and in fact against the principles of independence, equality and autonomy of reconfiguration, to allow an independent device to detect faults and communicate such events to core LRMs. For example, gateways could theoretically be equipped with software for event detection and communication. Thus, every processing module will require similar failure detection procedures. It is vital, that whatever detection technique is implemented in the system, it must not be susceptible to single message upsets, i.e. a mis-detection of a failure must not be caused by a single message being missed or misunderstood.

> Commentary:
> In order to be able to implement failure detection on data bus monitoring, it has to be required from the system that all the processing modules are at least fail-passive or fail-stop. [49] In such a situation the module will have to terminate all its communication on detecting an internal fault, and thus it will enable other core LRMs to detect the lack of its messages. The design of a fail-passive processing unit could be based on dual or multiple channel processing and comparison of results, or other fault detection and isolation techniques (FDI) which are widely discussed in literature (e.g. [41]).[50]

[49] Note that this requirement prevents the core modules from encountering arbitrary failures, otherwise called as Byzantine failures [39], [40].

[50] In the remaining part of this paper it will be assumed that all core LRMs are fail-passive or fail-stop.

In the design where handshake messages are used for signalling performance of avionics functions by particular processing modules, the failure of a module can be detected when the lack of a number of consecutive messages from this module has been noticed. A possible problem may occur in certain implementations, where a module will continue sending its handshake messages onto the data bus, whilst the performance of the avionics function has been aborted. In such a situtation the internal failure might not be detected as neither of the processing channels executes the application, hence no disagreement between channels will occur. If such a mode of failure is not extremely improbable, the failure detection techniques based on the function data monitoring should be preferred.

Failure detection approaches based on function data monitoring avoid the above identified problem. In this situation the loss of the function data can be strictly identified with the loss of the function, as even if a core LRM still performs the function but does not produce accessible results, it should be considered lost by the system. Again, depending on the standard and the communication protocol of the data bus, the task of matching the message with the function it has arrived from might be implemented based on the number of the transmission window, the label of the message, the destination address or other methods.

Commentary:
A very interesting modification of the ARINC 659 standard has been proposed in [42], that allows for very simple implementation of the failure detection mechanism based on the data bus access tables and dynamically modifiable module identity.

There may occur some latency problem related to the failure detection algorithms based on data monitoring. Different avionics functions access the data bus with different time intervals and thus the failure detection delay would have to be long enough to avoid a mis-detection of a loss of a "slow" function. On the other hand such long failure detection delays may not be acceptable for some "fast" and critical functions. A different solution could implement varying length failure detection delays depending on the particular avionics function. However, with failure detection delays varying from tens of milliseconds to hundreds of milliseconds or maybe even seconds, some combinations of failures, that were non-simultaneous for fixed delay algorithms, would now have to be considered nearly simultaneous (before a "slow" failure is detected a "fast" failure may occur). That would clearly lead to problems with the determinism of the reconfiguration scheme as discussed in section 5.2.3.

Unless the probability of an occurrence of nearly simultaneous failures and the severity of their consequences still meet the safety requirements - despite the varying failure detection delay - failure detection algorithms combining both techniques (handshake and data monitoring) are expected to be employed in order to obtain robust and reliable failure detection.

In general, a loss of an avionics function can be detected if the time of either absence of the function data or absence of the module handshakes, was longer than some algorithm and implementation dependent threshold.

> Commentary:
> Note that in order to implement the failure detection mechanism based on message time-outs, the system must guarantee that any message sent by a core LRM will be delivered within a certain time interval. This does not require the system to provide fully reliable communication medium, but simply requires the system to exhibit some synchronicity of communications [40], [39].

#### 5.2.4.4. Failure related actions

When a failure of a core LRM occurs, each of the operating processing modules must be able to perform the following tasks:

- failure detection - a loss of any avionics function must be detectable by some unified means
- evaluation of the reconfiguration process activation conditions
- reconfiguration process triggering, if the activation conditions evaluated in the previous step were satisfied.

### 5.2.5. Recovery

The notion of recovery in RIMA systems is of somewhat different importance than the notion of failure. Every reconfigurable avionics system must be able to continue operation in the event of a module failure, and it must comply with safety requirements such as those defined in [37], [3] and [36]. Thus a reconfigurable system must be proven safe regardless whether or not the subsequent recovery of a previously failed module will be allowed.

However, in many cases the availability of a system can be significantly increased should the system employ methods for assigning an avionics function to a module recovering from a transient failure. As previously mentioned in section 5.2.4.2, the capacity for module recovery allows the system to deal with soft failures. Moreover, it is believed that soft recoverable failures are more common to avionics systems than non-recoverable hard failures, thus reconfiguration schemes permitting a recovered module to select and perform an avionics function are very attractive.

It is also possible that early in the service life of an aircraft, the MTBF of particular units may be lower than assumed due to occurrences of soft failures. In such situations the benefits following system capacity for module recovery should not be underestimated.

#### 5.2.5.1. Definition

The notion of recovery in RIMA systems describes situations where a processing module passes its testing procedures and restores its operational capacity after suffering a temporary failure.

#### 5.2.5.2. Principles

As mentioned in the previous sections, the reconfiguration scheme must comply with safety requirements when assessed without recovery, as it is unacceptable to conduct a safety analysis based on the assumption that module recovery occurs with certain probability. Thus, the capacity for a module recovery has to be perceived as an additional (non-essential) quality of a reconfiguration scheme. However, as the soft failures are believed to occur more often than the hard ones, a reconfigurable avionics system implementing module recovery will exhibit greater endurance and higher availability than reconfigurable systems without any means for tolerating transient faults.

In some situations the processes of reconfiguration and recovery may want to alter the same reconfiguration data, thus all such indirect interactions must be proven to have no adverse effect on the reconfiguration scheme, and they must not lead to a violation of conditions and principles stated in section 5.2 and its subsections, nor can they breach derived requirements. Reconfiguration schemes should be designed with an attempt to eliminate any interaction between reconfiguration and recovery processes, for example by implementing recovery into standard module initialisation routines.

Commentary:
The implementation of the recovery and module initialisation procedures in a uniform manner, and a proof that the design of such routines is consistent with the design of the reconfiguration algorithm and its strategy data, renders the recovery of a processing unit transparent to the scheme at no additional cost (the recovering module simply undergoes its power-up programme).

Furthermore, the reconfiguration and recovery processes running on separate core LRMs must not interfere in the sense that a recovery of a processing module must not lead to a reconfiguration of another core LRM. Clearly, other interactions may exist, as all modules may have to detect another module recovery, and depending on the reconfiguration algorithm they may want to update certain reconfiguration data.

Some problems relate to possible inconsistencies in the reconfiguration data when a module recovers after a soft failure. If the reconfiguration algorithm utilises some dynamically updated data structures to determine the reconfiguration strategy (see sections 5.2.1.4 and 5.2.1.6), this data is likely to be out of date and inconsistent with the rest of the cabinet for a module that suffered a transient failure. The algorithm for reconfiguration and recovery must be capable of solving this problem by some consistency maintenance policy (see section 5.2.1.6) or by being insensitive to invalid data. Note, that if the latter approach is chosen, the reconfiguration algorithm must be fully functional for a module with partial or incoherent data, or alternatively the system must be able to tolerate a failure to reconfigure correctly by one or more modules, with a certain probability of occurrence of such an event.

As maintaining consistency of dynamically updated data increases the algorithm complexity, the reconfiguration and recovery methods based on static reconfiguration strategy data could be of some preference. Moreover, in the case of static reconfiguration strategy data actions performed by a recovering module can be left transparent to other core LRMs, and no recovery detection techniques will be required (processing module recovery procedures can be strictly confined to a single core LRM).

Some thought has to be given to the problem of the function selection algorithm. When a module restores its computational capacity it has to select an avionics function to perform. Clearly, only the functions that are not being performed at the time can be considered, as it is undesirable for two core LRMs to perform

the same avionics function[51]. The module must select the most critical function that is not being performed and for which it has the software or it is capable of downloading within the time constraints.

#### 5.2.5.3. Detection

A recovery of a processing module should be detected when data or handshake messages related to an avionics function that was previously lost appear on the backplane bus, or if an explicit recovery message is sent by the recovered module to all core LRMs. The latter method is of some preference as it avoids some problems related to timing.

> Commentary:
> In the case where the monitoring of the activity on the data bus is used in order to determine a module recovery some timing problems can be easily encountered. When a function is lost due to a module failure and an immediate backup for this function fails as well, all other modules could perceive the situation as a permanent function loss. When subsequently the second backup module takes over the avionics function, it could easily be mistaken for a module recovery and thus lead to reconfiguration process inconsistency or even failure. Clearly, a reconfiguration scheme can be designed to avoid or solve these problems without the need for explicit recovery indication[52], however, practice shows that event related communication improves the algorithm robustness.

#### 5.2.5.4. Recovery related actions

When a module suffers a soft failure it should perform the following tasks:

- termination of all data bus activity
- activation of self-test procedures
- if these have been successfully completed, the new function selection algorithm must be activated
- software for the selected function must be downloaded or fetched from the module non-volatile memory
- the recovery message (if any) should be sent on restoration of the function performance.

After all the previous steps have been completed, all of the operational core LRMs in the cabinet should perform the following actions:

---

[51] It is possible that in some cases two processing modules in separate cabinets may be required to perform identical applications in order to provide "hot" backup for the most critical functions.

[52] Certain algorithms based on static (never updated) reconfiguration data can operate properly without explicit recovery indication.

- if required by the reconfiguration and recovery algorithm, detect the recovery and update the relevant reconfiguration data.

### 5.2.6. Software downloading

Depending on the RIMA design a core LRM can:

- store the software for all the functions it may be requested to perform in its non-volatile memory
- store internally the software for some of the functions, with the necessity of dynamic downloading of other applications
- or it can thoroughly depend on software downloading from application modules

The criticality of the software downloading bus depends on the chosen approach as well as the reconfiguration algorithm.

In the situation where with every reconfiguration the necessary software has to be fetched from the application modules, the criticality of the software downloading bus corresponds to that of the most critical function within the cabinet, i.e. the probability of the loss of the SDB and a critical function module must be lower than $10^{-9}$ per flight hour. However, when core LRMs are capable of storing some avionics functions software, it is possible to reduce the criticality of the SDB. In such a case the SDB criticality corresponds to the criticality of the most important avionics function whose loss would require software downloading.

> Commentary:
> It is possible to design reconfiguration schemes where the least critical modules store software for the most critical functions, and thus two or more failures could be withstood without a need for the downloading of critical function software. In the case of a non-critical module failure the software could be fetched from another core LRM or from an application module, while in the case of a critical function loss the reconfiguration process could be successfully completed without software downloading.

It should be a design principle, that the software for the most critical avionics functions should be stored by the least critical modules, as this generally leads to shorter reconfiguration chains.

Several problems relate to the software downloading process:

- software request handling, i.e. how and to which module(s) should a core LRM indicate the need for the software
- software request response, i.e. how and which module(s) should answer to the software downloading request
- software source selection and software fetching, i.e. which module(s) should deliver the software and how should it be handled by the requesting core LRM.

On detecting the need for software downloading the core LRM should inform all possible software sources, for example by multicasting a software request message, in order to be able to select the module most appropriate for this purpose at the time.

> Commentary:
> It is undesirable for a processing module to attempt to contact only selected sources as this reduces the robustness of the reconfiguration algorithm (e.g. the software sources assigned to a particular module may have failed). It is, however, up to the particular implementation whether the core LRM contacts all the sources at once (broadcasting or multicasting) or on one-by-one basis (e.g. the processing module may attempt to contact the source $s_1$ and if this does not respond try $s_2$, $s_3$,..., $s_n$ until the appropriate response is received). It is expected that schemes based on contacting multiple software sources simultaneously will lead to shorter reconfiguration delays, and thus such solutions should be preferred.

Messages related to software request and software delivery can be exchanged on either the backplane bus or on the software downloading bus. As previously mentioned, the criticality of the SDB must match the criticality of the most important function for which it may be used, and as such it should be suitable for message exchange related to software downloading. Moreover, the use of the SDB for these purposes avoids an increase of the data traffic on the backplane bus.

On receiving a software request message a software source (a core LRM in RIMA architecture "C" or an application module in RIMA architecture "D") must check whether the appropriate software is available. Should this be the case, depending on the software downloading algorithm the module could either

- send a response message to the core LRM indicating its willingness to deliver the software (the processing module can than request the software directly from this source),
- or it could attempt to send the required software to the processing module.

The first approach gives the processing module a choice of the software source from all the modules that responded to the software request message. This approach avoids problems with data bus acquiring, which is inherent to the latter one, where modules compete for access to the shared resource. The second approach suffers also from problems related to multiple software delivery, i.e. a core LRM downloads the software from the fastest source, but other sources still attempt to deliver the software. If the software downloading bus and the algorithm are designed to withstand such problems, the second approach gives the benefit of shorter software downloading delays, as fewer messages need to be exchanged. However, as the speed of the SDB must be higher than that of ARINC 659 (see discussion in Chapter 1), the additional delay in the first approach will generally be negligible. Thus, due to its simplicity and robustness, the first approach seems to be of some preference.

### 5.2.7. Algorithm corruption

Although every effort has to be made in order to eliminate possible sources of algorithm corruption, the reconfiguration scheme (including the recovery algorithm) must comply with some general principles to guarantee acceptable system behaviour when reconfiguration data corruption occurs.

#### 5.2.7.1. Causes of data corruption

The two main classes of source of the data corruption can be identified:

- external to the method - sources that are difficult or impossible to eliminate, that are related to hardware faults or natural phenomena such as neutrino bombardment
- internal to the method - sources that are inherent to the method, for example, data corruption related to a mis-detection of an event or to undetected design or implementation errors.

It is believed that although every effort has to be made in order to protect the hardware from the external sources of data corruption, it is generally impossible to guarantee that the system is fully shielded from such undesirable interference. When the second class of data corruption sources is considered, the algorithm should comply with the following guidelines in order to eliminate the risk of inherent data corruption:

- detected failure or recovery events cannot be ignored, i.e. an operating mode in which a processing module ignores signals from the failure detection algorithm is not allowed

Commentary:
Some reconfiguration algorithms could allow the most critical functions to ignore failure and recovery events related to other modules. Such behaviour may in the case of some hardware or software problems result in a non-critical function acquiring the "ignore events" mode of operation, and its subsequent elimination from the reconfiguration scheme.

- data consistency procedures, such as those discussed in section 5.2.1.6, when properly implemented, lead generally to a reduced risk of data corruption
- algorithms avoiding dynamic reconfiguration strategy data updates are generally insensitive to algorithm inherent data corruption, unless implementation or design errors lead to data mis-handling.

Since the reconfiguration data is algorithm specific, every reconfiguration method should conform to the data handling guidelines, as well as identifying and preventing problems specific to the method.

#### 5.2.7.2. Required reconfiguration process behaviour in the event of data corruption

Regardless of the source and the actual data corruption, the reconfiguration algorithm must exhibit the following properties:

- reconfiguration process activation conditions (see section 5.2.1.4) must be satisfied before the actual reconfiguration can be triggered
- the failure detection mechanism should be able to operate, although its time constraints and delays could be invalidated
- failure related actions must be taken on failure detection (see section 5.2.4.4)
- corrupted data may lead to invalid activation of the reconfiguration process but must not prevent reconfiguration when activation conditions are satisfied
- data corruption must be contained within the affected module and must not propagate through the system.

#### 5.2.7.3. Required recovery process behaviour in the event of data corruption

When appropriately designed the recovery process should not be essential to the system safety, thus its operation in the case of reconfiguration data corruption should be allowed to lead to the following situations:

- duplication of functions in the cabinet – it is not critical if two identical avionics functions are performed by different core LRMs provided that the smart actuators or remote data concentrators are able to select their preferred data source
- lack of recovery – it is not critical if a module does not recover to perform an avionics function, as the system should have been previously operating safely on a degraded level.

### 5.2.8. Fault indication - warning messages

A strict relationship between processing units and avionics functions can be observed in non-reconfigurable avionics systems. In the case of a module failure it is relatively easy to inform the crew which function has been affected, either in terms of reduced redundancy or in terms of system degradation.

In RIMA systems a failure of a processing module will usually affect more than just one avionics function. Clearly, the function performed so far by the failed module will be affected at least temporarily, and in the case of a critical function failure, some other function may also have to be reconfigured should the redundant modules be not available. Moreover, in the case of longer reconfiguration chains even more avionics functions could be temporarily affected by a single core LRM failure. Therefore, fault indication for the crew should reflect stable states of the cabinet, i.e. a loss of an avionics function should be pronounced after the reconfiguration process has been completed. This will impose some new requirements on the flight warning system and will generally lead to a small delay in fault communication.

Since the reconfiguration delays must be proven to be shorter than allowed function absence time (section 5.2.2.1), the temporary loss of some function due to reconfiguration should be transparent to the crew. Moreover, as dynamic in flight reconfiguration must be fast, the time between a failure of a physical device and completion of the reconfiguration process can be expected to be shorter than one second, that should render the delay in fault indication negligible.

At the termination of the reconfiguration process, data related to avionics functions performed by particular core LRMs should be updated and the state of the system displayed to the crew. In the case of

a temporary loss of an avionics function that requires immediate corrective action from the crew[53], the failure occurrence and the activation of the reconfiguration process should be indicated to the crew when the fault is detected. Since reconfiguration is likely to finish in a very short period of time, such information will be of little benefit in other cases.

The system summary should be displayed to the crew when a stable state of the cabinet has been reached, including the information about the failed module and the lost function. Informing the crew about the reconfiguration path is undesirable, as it would not bring any relevant information and could cause various problems related to receiving a great amount of unnecessary data.

It seems inevitable in RIMA systems, that in order to achieve meaningful communication between an autonomous cabinet and the crew, some delays have to be inflicted on the warning indication procedures.

### 5.2.9. Function state updates

Computation of some of the avionics functions (e.g. navigation) can be based on the function state. As in RIMA systems there are multiple copies of avionics function software, there may occur some problems with preservation of the function state coherency.

Generally, every software copy of a function requiring state should be stored with its state, regardless whether the copy is stored on a core LRM or on an application module. The processing module responsible for current computation of the function should send the state information to the backplane data bus or it could be alternatively propagated on the software downloading bus, should its criticality allow such operation.

Some consideration should be given to the problem of increased data traffic related to the state updating policy via the backplane bus. As continuous updating of the function state could easily lead to a considerable increase of the data bus traffic, state preservation should be understood in terms of

[53] Such situations are unlikely as reconfiguration delays are shorter than the allowed function absence times.

checkpoint saving, where in the case of a failure the required application is re-executed from its last saved state checkpoint.

The time intervals between consecutive state updates and the size of the data items representing the state depend on the particular avionics function. It is expected that the state update rate will be much lower than the data transmission rate for every avionics function within the system, i.e. the state will not be updated with every result. Moreover, it is likely that the data items forming the state of an avionics function will be small, and thus they could be contained within a single message. Taking into account the two above arguments, it could be expected that the data traffic overhead related to state updates will be relatively small. This problem requires, however, some further analysis in order to confirm that the communication overhead will be acceptable.

An alternative method of function state preservation is proposed in [23], that requires the operating system to store an abstract representation of the function state, that in the case of an application failure could be transferred to another processing module. However, such an approach seems to be highly unfeasible for RIMA systems, as it is able to tolerate only a very small class of failures - failures that affect the application software but do not affect the operating system and the data bus interface.

The issue of preservation of application state in real-time and distributed systems has also been discussed in various papers referred to in Chapter 3.

## 5.3. Reconfiguration Algorithm Design Guidelines

In section 5.2 various aspects of dynamic autonomous reconfiguration schemes were discussed, possible problems were identified and some solutions were suggested. In this section preferred attributes and an outline of a reconfiguration algorithm are proposed. Note, that this section should be understood as design guidelines only, and that all the solutions and choices presented are not definitive.

The following table (Table 5.1) presents preferences in approaching different aspects of dynamic autonomous reconfiguration as discussed in section 5.2 and its subsections.

| Domain | Preferences | Section(s) |
|---|---|---|
| Properties | "normal" (possibly "extended") determinism, integrity | 2.3 |
| Time delays | short reconfiguration delays - no software downloading bus | 2.2.1 |
| Time delays | short reconfiguration chains | 2.2.2 |
| Communication | implicit exchange of information | 2.1.1 |
| Synchronisation | implicit multiple messages based phase-synchronisation | 2.1.2 |
| Failure | capacity for withstanding soft and hard failures (possibly with provisions for operation in case of latent failures) | 2.4.1, 2.4.2 |
| Failure detection | based on data monitoring (possibly combined with handshake messages) | 2.4.3 |
| Recovery | capacity for non-communication based autonomous recovery from soft failures | 2.5.1, 2.5.2, 2.5.3 |
| Recovery detection | autonomous and specific to recovering module (possibly with explicit post-recovery messages) | 2.5.3 |
| Functions software storage | on-board non-volatile memory of core LRMs (possibly further combined with some software downloading strategy) | 2.2.1, 2.6 |
| Reconfiguration data | static (no updates) reconfiguration strategy data | 2.1.6 |

**Table 5.1. Desirable attributes of reconfiguration algorithms.**

It is expected that the reconfiguration software will be running concurrently with the execution of an avionics function, or it may be integrated with the application software. Since the core LRM operating system or application executive will provide multitasking functionality [10], the implementation of concurrent execution of the reconfiguration process and the avionics function is expected to be relatively straightforward with the first approach. The latter solutions appear to constitute the less attractive choice, as it will probably require some modifications to the already existing avionics application software. Also, since the criticality of the reconfiguration software will be higher than that of non-critical functions, the integration of the reconfiguration process and particular avionics functions will lead to a significant and unnecessary increase of the software criticality, complexity, integrity and its cost of development and maintenance.

A reconfiguration algorithm could encompass the following functional blocks:

- Initialisation and Recovery Block

- Data Bus Monitoring and Message Recording Block
- Reconfiguration Block.

The following table (Table 5.2) shows the expected functionality of the above mentioned blocks.

| Block name | Functionality |
|---|---|
| Initialisation and Recovery Block | initialisation of auxiliary reconfiguration data required for further operation of the reconfiguration algorithm<br>initial monitoring of the backplane bus and function selection uniform for system start-up and module recovery (section 5.2.5) |
| Data Bus Monitoring and Message Recording Block | monitoring the backplane bus for messages arriving from other modules/functions (section 5.2.1.1, 5.2.4 and 5.2.5)<br>updating the auxiliary reconfiguration data to reflect the received information with respect to the current state of the cabinet (handshake/data message) and the applications state (section 5.2.9) |
| Reconfiguration Block | evaluation of the reconfiguration activation conditions (section 5.2.1.4)<br>should the conditions be satisfied, termination of current function, software fetching/downloading (section 5.2.6) and function re-execution<br>if required, updating of the reconfiguration strategy data and the auxiliary data |

**Table 5.2. Design blocks of the reconfiguration algorithm and their functionality.**

Some consideration has to be given to tasks such as termination and re-execution of an avionics function by the reconfiguration process. As the reconfiguration and application processes are expected to be independently running in a multitasking environment, there have to be made some provisions for the reconfiguration process to issue termination and activation requests. Such provisions could, for example, be handled with the use of events (as discussed in section 2.3.6.2.2 of [10]) or other inter-process communication methods (see section 2.3.6 of [10]). Alternatively, the functionality of process termination and activation could be embedded into the operating system as system calls or system services.

The following figure (Figure 5.3) shows the flow diagram of a possible reconfiguration algorithm.

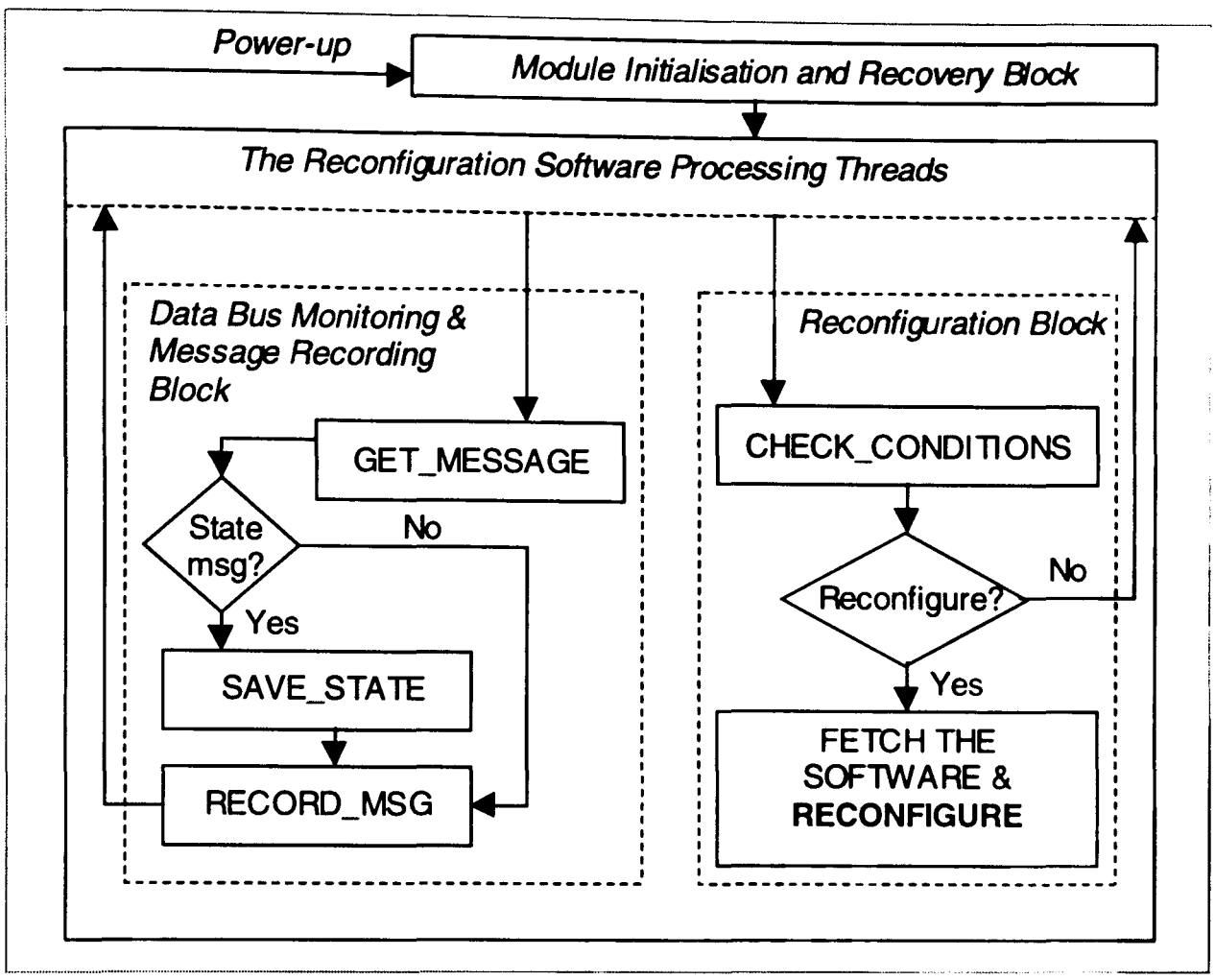



**Figure 5.3. Flow chart of a reconfiguration algorithm.**

It is possible that a processing module may not be able to select an avionics function to recover to. Such a possibility relates to the fact that in the case of the first few failures it will be the redundant core LRM that will reconfigure. Therefore all the cabinet avionics functions will be performed and the recovering module will not be able to select a lost function, thus continuous attempts to select an avionics function to recover to, implement indirectly the functionality of a redundant processing module.

Commentary:
Although the implicit implementation of a redundant module follows naturally the design of the algorithm it could lead to some problems with the determinism of recovery, even if the events occur in long time intervals. In the situation where two previously lost modules pass their self-test procedures and attempt to recover, and there are no lost functions as the redundant modules have reconfigured to sustain them, they will both assume that they are now providing the immediate backup for all functions and will reconfigure at the same time should another function be lost. This can be easily avoided if the recovering module selects a "redundant" function should other avionics applications be performed by the cabinet, that will place it at a definite position within the reconfiguration scheme. The "redundant" functions would have to manifest their performance by sending appropriate messages onto the data bus.

Note, that the above considerations on reconfiguration algorithms are intentionally free from any implementation specific details in order to avoid bias towards particular solutions.

## 5.4. Conclusions

Various aspects and key issues in designing autonomous dynamic reconfiguration schemes have been discussed in this chapter. Problems related to both the design and implementation of different reconfiguration tasks have been identified and possible solutions have been outlined. Finally, the algorithm design guidelines were proposed, that have been used in subsequent phases of the research to construct various reconfiguration methods applicable to RIMA systems.

At this stage any bias towards particular implementations was intentionally avoided in order to provide most generic discussion. On the other hand some design choices have been made (e.g. failure detection based on data bus monitoring or implicit phase-synchronisation), as it is believed that they should lead to a reduction in the complexity of the resultant reconfiguration software and to more straightforward implementation (hence lower probability of the occurrence of design and implementation errors as a possible safety benefit).

# Chapter 6. Autonomous Dynamic Reconfiguration Schemes

## 6.1. Introduction

Reconfigurable Integrated Modular Avionics (RIMA) employ dynamic reconfiguration in order to sustain the critical and essential functions of the system. It has been shown in Chapter 4 that such avionics systems can operate with significantly lower processing modules redundancy and still can greatly exceed the availability and reliability levels of traditional non-reconfigurable avionics. As the reconfiguration schemes become essential for preservation of critical functions in RIMA systems, they have to be considered safety-critical and as such they must comply with certain safety related requirements. These have been identified and discussed in Chapter 5, and recommendations for design of autonomous dynamic reconfiguration schemes have also been given.

In Chapter 3, the existing reconfiguration methods were analysed with respect to their applicability to dynamic reconfiguration of avionics systems. It has been found, that none of the reviewed methods can be directly employed to reconfiguration of RIMA, although parts of particular solutions can be adapted to implement different aspects of required reconfiguration algorithms.

In this chapter, various autonomous dynamic reconfiguration schemes are proposed and their properties are discussed. Two dedicated software models of RIMA architecture "C" and "D" had been developed to practically verify various properties of the schemes. Each of the reconfiguration schemes discussed in this chapter had been implemented and tested on a University of Bristol mainframe server. This allowed to draw various conclusions related both to the complexity of the code required to implement the schemes and to the properties of the schemes themselves. Selected schemes had been subsequently implemented on a more representative system (see Chapter 9 for details), indicating that presented methods can be successfully implemented into Reconfigurable Integrated Modular Avionics.

The remainder of the chapter is organised as follows. In section 6.2 the main aspects of reconfiguration schemes are discussed, including fault detection mechanisms, reconfiguration data and reconfiguration algorithms, and module recovery. In section 6.3, proposed autonomous dynamic reconfiguration schemes

are presented and compared. Their applicability to real-time and safety-critical systems is discussed, and the paper is concluded in section 6.4.

## 6.2. Main aspects of reconfiguration in RIMA

As already discussed in Chapter 5, the reconfiguration schemes applicable to implementation into RIMA should be autonomous (should not rely on any executive module(s)) and dynamic (should allow for reconfiguration during system operation). Based on such requirements it is expected that the employed scheme(s) will encompass the following phases:

- fault detection - processing modules will detect failure of other core LRMs by monitoring for loss of data on the backplane bus,
- reconfiguration - when a failure is detected, the module will decide whether or not to reconfigure, based on stored reconfiguration strategy data,
- initialisation and recovery – on system power-up or when recovering from a transient failure the module will select and execute appropriate function, or it will act as a redundant module.

Optionally the scheme may employ some software downloading mechanisms, should the required software be not available directly from the module non-volatile memory. However, as software downloading from an external source will in general lead to significantly longer reconfiguration delays, solutions based on locally stored applications are preferable.

Clearly, there can be very many implementations of the above mentioned phases that would comply with the requirements as defined in Chapter 5. In the following sections some of the possible solutions are presented, discussed and appropriate recommendations are made.

### 6.2.1. Fault detection

It has been discussed in Chapter 5, that failure detection mechanisms are almost inevitably to be based on backplane bus monitoring. From the reconfiguration software point of view there is no difference between monitoring application related data or specific handshake messages, provided that each message allows for identification of its source, i.e. the application that has produced it. With such an approach the

reconfiguration software will be able to monitor for any backplane bus activity from particular avionics functions.

In order to avoid single message upsets the failure detection mechanism should announce a loss of an avionics function only if at least $K$ consecutive messages related to the function are lost. The factor $K$ will depend on the allowed function absence time, the rate at which the function produces its messages and the required reliability of the failure detection mechanism (the greater the $K$ the lower the probability of invalid failure detection).

### 6.2.2. Reconfiguration algorithm and reconfiguration data

In the event of a successful failure detection the reconfiguration routines will be called. Based on the current state of the cabinet and the reconfiguration data, the reconfiguration algorithm determines if the module should react and what actions it should undertake (i.e. should the module reconfigure or should it only note the failure). The behavioural requirements and desirable properties of reconfiguration algorithms have been discussed in Chapter 5, at this point the considerations will be focused on practical implications following certain implementation choices.

As outlined in Chapter 3 and further discussed in Chapter 5, reconfiguration algorithms based on pre-generated data will in general lead to short reconfiguration delays (no need for complex computational tasks), and possibly to very high levels of determinism and predictability. All the data items that are necessary for real-time operation can be generated during the system design and implementation phases, and their properties can be relatively easily proven or tested. Therefore, the reconfiguration algorithms could be designed to lookup all their decisions in reconfiguration data tables, that would generally lead to very simple reconfiguration algorithms, and thus to very a low probability of implementation or design errors.

In general two alternative approaches to operation with the reconfiguration data can be identified:

- not to change any data items during operation and only use the data for reference (static data)
- to dynamically update the data to reflect the changes of the system state.

In the first approach, the properties of the data, and thus the properties of the reconfiguration scheme, are preserved at all times and the behaviour of the system can be easily predicted. Also, all of the processing modules and their reconfiguration processes will reference the same data, which will reduce the possibility of inconsistent or incoherent actions.

> Commentary:
> There may arise a situation where the reconfiguration data will become inconsistent due to some data corruption, e.g. related to some spontaneous change of memory location or neutrino bombardment. Issues related to data corruption and data consistency maintenance have been discussed in Chapter 5.

In the case of dynamically updated reconfiguration data, the possibility of discrepancies between different core LRMs is greatly increased and the system behaviour could prove to be more difficult to predict. On the other hand, algorithms based on dynamic data can adapt to the continually changing system state, and are theoretically capable of operating more efficiently and with shorter delays. Therefore, the choice between static and dynamic reconfiguration data can be understood as a choice between simple and less failure prone schemes and faster but more complex ones.

### 6.2.3. Recovery

Module recovery provides good means for tolerating transient faults. The certification and safety issues inherent to the situation where a previously failed module restores performance of an avionics function have been discussed in detail in Chapter 5. However, there may be some practical problems related to the implementation of recovery into a reconfiguration algorithm.

After being off line for a period of time some of the module reconfiguration data will have to be considered obsolete, regardless of whether the module was able to preserve the state from before the failure or not. Thus, all the reconfiguration related data of the recovering module will require initialisation without causing undesirable interference with other core LRMs. In the case of the failure detection mechanism it could be simply assumed that messages from all functions were present at the last check, and only the lack of the following $K$ messages will indicate the function loss. However, the situation could become more difficult when validity of the dynamic strategy data is being considered.

Commentary:
In the case where the module has to update its strategy data as the system encounters failure and recovery events, a module taken off line for a period of time due to a failure will have to consider its data obsolete (multiple events might have occurred in the meantime, that have not been recorded). Clearly, reconfiguration schemes employing static strategy data will not encounter such a problem.

An obvious solution would be to request reconfiguration data from all the working core LRMs, and then perform some form of majority voting or adjudication. However, such a solution could lead to somewhat increased traffic on the backplane bus, and possibly to greatly increased complexity of the reconfiguration software. Alternatively, the recovering module may attempt to recreate the valid strategy data based on the current state of the cabinet exploiting scheme determinism (e.g. if the contents of the strategy table depends only on the number of working core LRMs, the recovering module will be able to recreate it based on backplane bus monitoring).

The above arguments suggest, that the reconfiguration algorithms based on static reconfiguration data provide a good basis to allow module recovery without an undesirable increase of software complexity. In the case of algorithms based on dynamically updated reconfiguration data, the capacity for module recovery will in general lead to somewhat increased complexity and an increased risk of design and implementation errors.

## 6.3. Proposed schemes

All reconfiguration schemes presented in this section follow the same general design, which has been presented and discussed in Chapter 5 (see Figure 5.3). The differences between schemes refer mostly to the actual reconfiguration strategy and the reconfiguration data, although some minor alterations to the failure detection and the recovery mechanisms were also required to preserve consistency within particular reconfiguration schemes.

The implementation of the application termination and re-execution routines can be expected to be identical for all schemes and it will depend on the available operating system (OS) and application executive APEX [10]. To allow the core LRM to execute an application from a previously saved state,

some alterations to the application software will be inevitable despite the actual separation of the reconfiguration process and the avionics function.

Also, the failure detection mechanism can be designed and implemented with a high level of commonality between different schemes. In general, the failure detection routine records in an auxiliary table the times of latest responses from all avionics functions, where the notion of a function response accounts for any data bus activity originating from the avionics function being considered. With every received message, the difference between the time of the latest activity from all functions and the present time is calculated, and in the case of "too old" responses[54] the function loss is detected, and the module calls its reconfiguration routines to determine its actions.

During initialisation or recovery a core LRM selects the most critical function that is not being performed and for which the module has got or is able to obtain the relevant software. As the core of the function selection mechanism is also based on backplane bus monitoring for activity from all avionics functions, in some implementations the failure detection and function selection routines could be partially integrated or they could call common subroutines.

Although particular reconfiguration algorithms that are presented in the following sections differ, they all utilise similar reconfiguration data but provide unique data handling procedures. The general approach is to implement the reconfiguration strategy into a look-up table similar to that shown in Figure 6.1 below, where the non-negative sub-diagonal matrix entries correspond to backup levels for particular functions (each avionics function is represented as a column), and the non-negative super-diagonal entries represent software availability on module recovery. All the negative values describe the situation where the relevant module is not in backup for the function.

[54] A predefined absence interval referred to as FAIL_DELAY, accounts for the loss of several consecutive messages from an avionics function, and may differ between functions.

| Module | | Function | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | | C | H1 | H2 | MJ | M1 | M2 | R | R |
| 0 | Catastrophic | **-1** | -1 | -1 | 3 | 2 | 2 | -1 | -1 |
| 1 | Hazardous | 3 | **-1** | -1 | 4 | 3 | -1 | -1 | -1 |
| 2 | Hazardous | 2 | -1 | **-1** | 5 | -1 | 3 | -1 | -1 |
| 3 | Major | 1 | 3 | 3 | **-1** | -1 | -1 | -1 | -1 |
| 4 | Minor | 0 | 2 | 2 | -1 | **-1** | -1 | -1 | -1 |
| 5 | Minor | -1 | 1 | 1 | 2 | -1 | **-1** | -1 | -1 |
| 6 | Redundant | -1 | 0 | -1 | 1 | 1 | 1 | **-1** | -1 |
| 7 | Redundant | -1 | -1 | 0 | 0 | 0 | 0 | -1 | **-1** |

**Figure 6.1. Example of a module based reconfiguration strategy look-up table.**

The ways core LRMs use the look-up table to determine their reconfiguration policies vary from one reconfiguration scheme to another, and are discussed later in this chapter. Also different approaches to the design and implementation of other functional blocks are discussed in the following sections, where particular reconfiguration schemes are presented.

### 6.3.1. Static data based reconfiguration schemes

Two alternative schemes are discussed in this section, that operate with no updates to the reconfiguration data. In the first of them the reconfiguration actions are bound to the module as a physical device, in the second approach it is the module function that is of greatest importance to the module reconfiguration policy.

#### 6.3.1.1. Module based look-up table approach

The module based look-up table could be identical to that in Figure 6.1. Each matrix row corresponds to a processing module and each column describes an avionics function. At the start of operation each module is assigned an avionics function based on its position in the cabinet, i.e. module "0" - function "0", module "1" - function "1", etc.[55] The [*module*][*function*] entry describes the *module* backup level for the *function*, where the value of "0" denotes the immediate backup and the value of "-1" corresponds to the "not in backup" situation.

The function criticality decreases as the matrix indices increase, thus positive entries above the main diagonal refer only to software availability on module recovery. The number of positive entries in each row is limited by the module available on-board non-volatile memory, as solutions employing a software downloading bus for software fetching from an external source are unlikely to be optimal in time-critical applications, thus the relevant programs should be stored locally.

> Commentary:
> As the mode of failure where an avionics function has been lost but the reconfiguration software remains functional has to be considered, the "-1" values on the main diagonal prevent the modules to reconfigure on their own failures. However, should the module suffer a transient fault and then successfully pass its self-test procedures it may be allowed to recover to its original function[56] (on recovery, based on its own ID, the module treats the appropriate main diagonal entry as non-negative – the software for the primary assignment function is considered available).

The reconfiguration strategy table can be understood as a superposition[57] of two identically sized tables related to reconfiguration and recovery. The reconfiguration table utilises only the sub-diagonal entries in order to determine backup levels, and thus the order of reconfiguration. The recovery table uses all the entries, but distinguishes only two states: "-1" - function software unavailable and "not -1" - function software available. Solutions employing two separate tables, although possibly easier to understand, do not bring any design or implementation benefits, but introduce some extra complexity related to generation and maintenance of multiple tables. Therefore, schemes designed and implemented to employ both tables as separate data items will not be further discussed.

If lack of the *i-th* function activity is noticed to be longer than a predefined $FAIL\_DELAY_i$ (time delay for the *i-th* function accounting for the loss of $K$ messages), the level "0" module will reconfigure. Generally, the loss of the *i-th* function for a time longer than $N \times FAIL\_DELAY_i$ will cause the *(N-1)-th* backup module for the *i-th* function to reconfigure. Note, that the reconfiguration process activation

---

[55] This is not the only possible assignment (the method does not impose restrictions in this matter), however it leads to somewhat simpler look-up table generation and use.

[56] Whether or not such a behaviour is permissible will depend on the coverage and confidence of module self-test.

[57] With the exception of the main diagonal which is treated differently as explained in the commentary.

conditions as identified in Chapter 5 must still be satisfied, thus the module will never attempt to reconfigure if the lost function is not more critical than its current one.

If a previously failed module recovers, it reconfigures to the most critical function that is not being performed and for which it has a positive matrix entry (for all cabinet functions the lack of backplane bus activity for the predefined $RECOVERY\_DELAY_i$ determines the loss of the *i-th* function during the module recovery phase).

#### 6.3.1.2. Function based look-up table approach

The strategy look-up tables differ for module and function based approaches (see Figure 6.2), as in the latter method it is impossible to constrain the amount of non-volatile memory required by each processing module other than by dividing the cabinet into sub-cabinets or by implementation of software downloading via a data bus.

> Commentary:
> In schemes employing function based strategy tables a reconfigured module will be in identical backup levels for the same avionics functions as the lost module before failure. Therefore, if the redundant modules are in backup for non-critical functions, and non-critical core LRMs are in backup for essential and critical functions; after reconfiguring to a non-critical function the module may be later required to reconfigure to sustain some critical function, and thus it will require software for all the functions in the cabinet.

Both solutions allowing for reduction of the amount of the non-volatile memory would lead to significantly lower benefits of dynamic reconfiguration, in terms of either increased redundancy or increased reconfiguration delays. Therefore, in the remaining part of this Chapter it will be assumed that each of the core LRMs will have sufficient memory to store internally all the functions performed by the cabinet.

> Commentary:
> Memory arrangements necessary to implement methods employing function based look-up tables should be economically and physically feasible due to continuous decrease of non-volatile memory chips prices and increase of their capacity. Moreover, as the cabinets can be expected to perform between eight and ten avionics functions (see the analysis in Chapter 4), only some 8 to 10 MB of non-volatile memory will be required per core LRM, assuming the size of each function software to be in the order of 1 MB.

| Module | | Function | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | | C | H1 | H2 | MJ | M1 | M2 | R | R |
| 0 | Catastrophic | **-1** | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 1 | Hazardous | 6 | **-1** | -1 | -1 | -1 | -1 | -1 | -1 |
| 2 | Hazardous | 5 | -1 | **-1** | -1 | -1 | -1 | -1 | -1 |
| 3 | Major | 4 | 4 | 4 | **-1** | -1 | -1 | -1 | -1 |
| 4 | Minor | 3 | 3 | 3 | 3 | **-1** | -1 | -1 | -1 |
| 5 | Minor | 2 | 2 | 2 | 2 | -1 | **-1** | -1 | -1 |
| 6 | Redundant | 1 | 1 | 1 | 1 | 1 | 1 | **-1** | -1 |
| 7 | Redundant | 0 | 0 | 0 | 0 | 0 | 0 | -1 | **-1** |

**Figure 6.2. Example of a function based strategy look-up table.**

As in the previous method, each column in the strategy look-up table corresponds to backup levels for the appropriate avionics function, where the values of the particular entries should be understood as in the case of the module based look-up tables (see section 6.3.1.1). However, the matrix rows are not statically bound to processing modules, but on detection of the function loss each module determines which row of the table it is expected to look-up based on the function its is performing.

For example, if a redundant module "7" has previously reconfigured to the major function "3", in the case of the following detection of a function loss it will refer to the row "3" in the table shown in Figure 6.2. Having established the appropriate row, the reconfiguration routine makes its decisions identically as in the module based method, i.e. the loss of the *i-th* function for a time longer than $N \times FAIL\_DELAY_i$ will cause the *(N-1)-th* backup level module for the *i-th* function to reconfigure. Also, similarly as in the previous method, the reconfiguration process activation conditions must be satisfied prior to core LRM reconfiguration.

Note that the underlined entries in the above table provide additional fault tolerance beyond the requirements identified in Chapter 4. They are optional, and can be changed to "-1" without violating any safety or availability objectives.

In the case of a transient failure and following module recovery, as the software for all the cabinet functions is stored in each module non-volatile memory, the module will reconfigure to the most critical

function that is not being performed at the time. To identify the lost functions the module monitors the backplane bus for the lack of the *i-th* function activity for the predefined time interval of $RECOVERY\_DELAY_i$.

#### 6.3.1.3. Discussion and comparison of module and function based approaches

In the module based approach it is the physical position of the module (its ID) that determines the module behaviour in the case of failures. The capacity for reduction of the amount of non-volatile memory required for each of the core LRMs (within the limits imposed by the system safety and availability requirements) could be perceived as an advantage of this approach when compared to function based schemes. However, it can lead to sub-optimal behaviour with respect to long reconfiguration chains.

> Commentary:
> In the case of the module based strategy look-up tables where the module behaviour depends on its position in the cabinet, an originally redundant module that has reconfigured to perform an avionics function will again reconfigure first should a more critical function be lost. Thus, depending on its function criticality and the state of the system, the function executed by the originally redundant module forced to another reconfiguration may have to be undertaken by another processing unit, that will lead to longer reconfiguration chains.

On the other hand, function based strategies allow for the shortest possible reconfiguration chains always of the length of one. Regardless of the system history and its present state it is always the least critical function module that will reconfigure should the criticality of the lost function justify reconfiguration. Furthermore, in function based approaches the behaviour of the reconfiguration scheme depends only on the set of functions performed by the cabinet, and not on the assignment of avionics functions to particular core LRMs.

> Commentary:
> If module A reconfigures from function Y to function X, in the case of following failures it will look-up the reconfiguration strategy for function X and not Y (represented in the look-up table by a separate rows). Therefore, functionally and with respect to the reconfiguration scheme there will be no difference whether it is module A or module B that is performing function X, as actions undertaken by either of them will be identical.

Such a property of the reconfiguration scheme can be understood as the capacity for deterministic functional system degradation.

Commentary:
After a core LRM failure and the following reconfiguration, the system will be homomorphic with the system from before the failure with the least critical function module removed. Thus each processing module failure when combined with reconfiguration can be functionally modelled as a failure of the least critical module. With exception of some transitory behaviour during the actual reconfiguration, each failure can be represented as a failure of one of the least critical modules.

Such a feature could be potentially useful when certification of a reconfiguration scheme is being considered, as the complete functional evolution of the reconfiguration scheme can be easily predicted and analysed irrespectively of the number of possible combinations of failures.

The above arguments suggest, that the reconfiguration schemes employing static function based look-up tables offer much more desirable and deterministic behaviour, than those employing module based strategies. It also seems, that the necessity for extensive software store in the first approach should be easily affordable, and the obtained benefits would justify the cost of additional non-volatile memory. The advantages and disadvantages of particular approaches are summarised in the following Table 6.1.

| Module based reconfiguration strategy | Function based reconfiguration strategy |
|---|---|
| **Advantages** | |
| Allows for limitation of the necessary amount of non-volatile memory | Each failure involves reconfiguration only of the least critical function module (reconfiguration chains limited to a single module) |
| | Allows for deterministic functional system degradation |
| | Highly predictable |
| | Exhibits potential for certification with simplified procedures |
| **Disadvantages** | |
| Possibly sub-optimal reconfiguration chains (even longer than two) | Does not allow for limitation of the amount of module non-volatile memory without incurring a considerable redundancy overhead (sub-cabinets) |
| Possibly more complex and difficult certification with respect to functionally different modes of failure | |

**Table 6.1. Comparison of static data based reconfiguration schemes.**

### 6.3.2. Dynamic data based reconfiguration schemes

The dynamic data based reconfiguration algorithms update their reconfiguration strategy data as the system evolves. Such behaviour leads to a close equivalence between the reconfiguration scheme and the state of the cabinet, thus allowing the reconfiguration algorithm to optimise its actions. The biggest benefit following the implementation of this type of algorithm is expected to lie in significantly shorter reconfiguration delays in the case of multiple non-simultaneous failures[58].

However, when dynamic data based reconfiguration algorithms are being considered, some problems related to module recovery occur. After suffering a transient failure and being unable to monitor the state of the cabinet (i.e. being unable to update the relevant reconfiguration strategy data), the module has to perceive its data as obsolete. Appropriate procedures therefore need to be devised to allow the module to recreate its data or to make it coherent with other core LRMs. The approach adapted to the methods discussed in this chapter depends on the expected equivalence of recovery related actions between the recovering module and other core LRMs, that allows for data regeneration on module recovery (the procedures are explained to a greater detail later in this section). In this class of algorithms the complexity of the reconfiguration software can be expected to increase considerably, and the choice between the particular approaches may not be straightforward.

#### 6.3.2.1. Dynamic data based reconfiguration algorithms without recovery

The objective of the dynamic data based reconfiguration algorithms is to allow the shortest possible reconfiguration delays regardless of the state of the cabinet. This can be achieved with reconfiguration schemes based on the strategy look-up tables as described in the previous sections (see Figure 6.1 and Figure 6.2 for examples of module and function based strategy look-up tables), should the following properties of the tables are preserved:

- each cabinet function with a backup will have exactly one core LRM in the immediate backup mode (i.e. the appropriate entry in the strategy look-up table will be equal to "0"),

[58] It was discussed in the previous sections that in the case of static algorithms the delay required to trigger the reconfiguration process increases as higher backup levels are required to reconfigure.

- should there be multiple backup modules for function *F*, their backup levels (entries in the appropriate column) will be consecutive (i.e. "0", "1", "2", ...), and modules performing more critical functions will have greater entries in the strategy look-up table.

Tables shown in Figure 6.1 and Figure 6.2 meet the above objectives, however the difficulty lies in devising algorithms that will update the strategy look-up tables in a manner that will not invalidate them. It is believed that the reconfiguration algorithms employing function based strategy look-up tables with no requirements on core LRMs non-volatile memory can accomplish this task with considerably lower complexity then the module based schemes. Therefore, the function based methods will be presented first, and the following discussion will relate to extensions required to successfully implement similar reconfiguration algorithms engaging module based strategy look-up tables.

***Reconfiguration algorithms with function based strategy look-up tables***

In the function based approach the functionality of the design blocks from Figure 5.3 remains unchanged, and the data used in static function based methods can also be reused at this place. The only required changes refer to the reconfiguration algorithm and its interaction with the reconfiguration strategy data.

On detection of a function loss a processing module calls its reconfiguration routine to identify the actions it is supposed to be taking, the algorithm determines the row of the look-up table it should use based on the avionics function it is performing (as in the schemes discussed in section 6.3.1.2), but only if the module finds that the appropriate entry is equal to "0" it will attempt to reconfigure.

Each operating core LRM updates the reconfiguration strategy tables in the same manner, regardless of whether it is required to reconfigure or not. The way in which the data is updated depends on the criticality of the lost function and the state of the cabinet, that is contained implicitly in the reconfiguration table. The following two cases can be distinguished:

- there is a backup module for the lost function that will reconfigure
- the lost function is one of the least critical ones and will not be sustained.

In the first case the least critical function module will take over the lost function, and thus (as explained in section 6.3.1.2) the failure will be mapped into the loss of the least important function referring to the last non-negative row in the strategy look-up table. This corresponds to the situation where the module who had the relevant entries equal to "0" reconfigures to the lost function, and thus it starts referring to a different row in the reconfiguration table. Therefore, all entries in its original row must be changed to "-1" (not in backup, the non-critical function is lost), and all other positive entries in the table decreased by one (i.e. because the level "0" module has reconfigured, level "1" must become "0", level "2" will change to "1", etc. in order to comply with data objectives outlined in section 6.3.2.1). Both actions can be simply accomplished by decreasing all non-negative table entries by one. Note that in function based tables all non-negative entries within any row are identical.

In the second case none of the operating modules will reconfigure, and thus it may not be possible to map such a failure into the loss of the function represented by the last non-negative row in the table (e.g. should the matrix shown in Figure 6.2 be used, a failure of the module "6" will not lead to reconfiguration and it will not relate to the last non-negative row).

> Commentary:
> In order to avoid such a situation, criticality of each function could be adjusted for the purpose of reconfiguration so that no two functions are of identical importance (e.g. minor function F1 could be defined as more important than minor function F2, and thus F2 would be allowed to reconfigure to F1). This would simplify the algorithm, but could also lead to reconfiguration between function of identical criticality, that may not be desirable. This is further discussed in section 6.3.2.2.

In such a case all entries in the failed module row need to be reset to "-1" (not in backup), and values for other core LRMs must be updated to preserve consecutive numbering. This can be simply achieved if each column entries greater than the one corresponding to the lost function are decreased by one, and all other entries remain unchanged. Note that this method will not alter the "-1" entries as they cannot be greater than any other table entry.

To allow a core LRM to reconfigure the time of the last activity of the lost function is reset to the current time. If after following FAIL_DELAY there is still no activity from the function, the next backup level (if one exists) will attempt reconfiguration and the table will be updated once again. The lack of backup

for a detected lost function and all relevant row entries equal to -1 will indicate that no further updates are required to the strategy table, as the function has been permanently lost.

Since all core LRMs execute the same update procedures and they monitor the backplane bus in an identical manner, their data should remain consistent provided that the failure detection mechanism is sufficiently reliable. Algorithms following the above discussed design exhibit the capacity for deterministic functional system degradation similarly to the schemes presented in section 6.3.1.2. Moreover, in this approach the state of the reconfiguration strategy table does not depend on the order of encountered failures, but only on the set of avionics functions not being performed (this takes into account some pseudo-functions of the redundant modules, and can be further exploited to implement data regeneration procedures that will be activated on module recovery).

***Reconfiguration algorithms with module based strategy look-up tables***

Schemes employing dynamic module based (i.e. physical location orientated) reconfiguration data require more changes to the original design shown in Figure 5.3, than those presented above. In order to allow the reconfiguration data stored by the processing modules to correctly reflect the state of the cabinet, the functionality of the failure detection mechanism needs to be extended. In the approach presented in this section, all core LRMs monitor the backplane bus activity not only to detect function losses, but also to identify module failures and to record functions performed by the modules. This can be achieved by the use of appropriate handshake messages, e.g. "*function M is performed by module N*", which convey the information about the functions being performed, the operating modules and the current assignment of functions to modules.

In the case of a detected function loss each core LRM calls its reconfiguration routine to determine whether it should reconfigure. As in the previous method the appropriate table entry equal to "0" indicates that the module will reconfigure. However, unlike in the previous scheme each module always refers to the same row of the reconfiguration strategy table.

All processing modules identify the module that is supposed to reconfigure, and update the table accordingly:

- if none of the core LRMs is to reconfigure, no changes are made to the reconfiguration table (such a case relates to a failure of one the least critical modules and it is dealt with by a separate routine invoked upon a detection of a module failure and not on a function loss),
- otherwise, backup levels for appropriate functions are adjusted to comply with the requirements identified in section 6.3.2.1, the procedure for which is explained below.

The key issue in the update algorithm is to identify all the avionics functions for which the reconfiguring module will stop being in backup. Clearly, if a redundant core LRM reconfigures to a critical function it will cease to provide backup for non-critical or essential functions. Therefore, for all functions of criticality lower or equal to the function causing reconfiguration (including the reconfiguring one), each core LRM decreases by one all non-negative entries greater than those of the reconfiguring module (this reflect the elimination of one backup core LRM for less or equally critical functions). Furthermore, for the same avionics functions all entries in the reconfiguring module row must be reset to "–1" to correspond to the "not in backup" situation.

The above algorithm is not sufficient to preserve all the desirable properties of the reconfiguration data. It is obvious that a failed module ceases to be in backup for any of the avionics functions, and the strategy table will also need to be updated for this. For all columns the failed module had a non-negative entry (i.e. was in backup), the update algorithm decreases by one all the entries which are greater than that of the failed module, which in turn is reset to "-1". Such a procedure preserves the consecutive backup levels and ensures that they will start from "0". The figure below (Figure 6.3) gives an example of table updates, for a simple case where all functions are of different criticality ("0" - critical, "4" redundant), and function "1" is lost.

| | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 0 |
| 1 | 3 | -1 | -1 | -1 | -1 | 3 | -1 | -1 | -1 | -1 | 1 |
| 2 | 2 | 2 | -1 | -1 | -1 | 2 | 1 | -1 | -1 | -1 | 2 |
| 3 | 1 | 1 | 1 | -1 | -1 | 1 | 0 | 0 | -1 | -1 | 3 |
| 4 | 0 | 0 | 0 | 0 | -1 | 0 | -1 | -1 | -1 | -1 | 4 |
| | Tale prior to the update | | | | | Table after the update | | | | | |

**Figure 6.3. Example of strategy updates for module based look-up tables.**

Thus for each of the detected module failures and corresponding function losses the following two update steps are taken:

- on detected function loss - all modules identify the module to reconfigure and update the table accordingly (backup levels change due to reconfiguration of a core LRM),
- all modules update the reconfiguration strategy data on detected module failure (backup levels change due to a loss of a module).

It appears that the task of preservation of the required strategy table properties in the module orientated schemes requires a great deal more processing than in the case of function based schemes.

#### 6.3.2.2. Recovery in dynamic data based reconfiguration methods

The difficulty in introducing recovery into dynamic reconfiguration algorithms follows the possibility of data becoming obsolete due to the module being off line. After successful completion of the self-test procedures the module needs to synchronise its data with all other operating core LRMs before resuming execution of an avionics function.

As discussed earlier, the explicit data exchange with possible majority voting may not be desirable due to increased algorithm complexity and to additional load on the backplane bus. In the case of previously described dynamic data algorithms implemented into ten or twelve core LRM cabinets, all the required data can be contained within a single 10x10 or 12x12 byte array, thus the increase of the data bus traffic related to the exchange of strategy tables can be considered insignificant when compared with the backplane bus throughput of at least 30 Mb/sec (ARINC 659). However, the autonomy of core LRMs would have to be questioned if their recovery was to be based on the data fetched from other processing modules. It is also possible, that although the data exchange based approaches will not create problems on the backplane bus, their complexity may be much higher than the complexity of alternative methods, such as algorithms attempting to recreate the reconfiguration strategy tables based on backplane bus monitoring for activity from particular functions or modules.

In the case of function based look-up tables, their independence from the sequence of failures can be used for that purpose. A recovering module fetches from its non-volatile memory an original version of the reconfiguration data and then starts monitoring the data bus for a pre-defined period of time to identify lost functions. Having done so, for each of the lost functions (including pseudo-functions of redundant modules) it calls the reconfiguration routine that will update the strategy table accordingly, pretending that the module is performing the most critical function. After being updated for each of the lost functions, the strategy data will be identical to the data of the operating modules.

Having done so, the module selects the most critical of the lost functions that it will reconfigure to, and sends a dedicated message to inform other core LRMs about its recovery (e.g. "core *N* recovers to function *M*"). On receiving the recovery message all modules including the newly recovered one will update their strategy tables in the following manner:

- all relevant entries for the *M-th* row will be set to "0" for each function that the module is supposed to provide backup for (immediate backup level)
- all other non-negative entries will be increased by one.

The above procedure will assure compliance with the data requirements from section 6.3.2.1.

> Commentary:
> As the recovering module will perform the least critical function within the cabinet, it will be expected to provide immediate backup for the remaining functions performed within the cabinet. Various issues relating to the confidence in self-test procedures and to allowing module recovery in this manner are discussed later in this chapter.

However, this solution will produce correct results only for the first recovering module, and there is no guarantee that the data will be correct for the following recoveries. The problem will manifest itself if a table contained discontinuous columns in terms of a negative entry separating non-negative ones, that appear when a non-critical module with no backup fails. For example, both redundant modules fail and then minor function "4" is lost in the system described by the strategy table from Figure 6.2. If subsequently a module recovers to function "4" the table will look as in Figure 6.4 below.

| Module | | Function | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | | C | H1 | H2 | MJ | M1 | M2 | R | R |
| 0 | Catastrophic | **-1** | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 1 | Hazardous | -1 | **-1** | -1 | -1 | -1 | -1 | -1 | -1 |
| 2 | Hazardous | -1 | -1 | **-1** | -1 | -1 | -1 | -1 | -1 |
| 3 | Major | -1 | -1 | -1 | **-1** | -1 | -1 | -1 | -1 |
| 4 | Minor | 0 | 0 | 0 | -1 | **-1** | -1 | -1 | -1 |
| 5 | Minor | 1 | 1 | 1 | 1 | -1 | **-1** | -1 | -1 |
| 6 | Redundant | -1 | -1 | -1 | -1 | -1 | -1 | **-1** | -1 |
| 7 | Redundant | -1 | -1 | -1 | -1 | -1 | -1 | -1 | **-1** |

**Figure 6.4. A possible strategy table after one recovery.**

The next recovering module will have no means (other than direct communication) to detect that function "4" is in lower backup level than function "5". It would have assumed that row "4" contains ones and row '5" contains zeros, that would be the case if no module had recovered and both redundant modules had failed. Therefore, the two chains of events could not be distinguished without communication between modules.

Some changes can be introduced into the reconfiguration algorithm that would eliminate the problems of discontinuous columns. The algorithm would have to allow functions of same criticality to reconfigure to each other (e.g. in the above situation on failure of the minor function "4", the application"5" would reconfigure, see also commentary in section 6.3.2.1).

> Commentary:
> With such an approach all failures within the cabinet could be mapped into the loss of the least critical function represented by the lowest non-negative matrix row. This would lead to even more deterministic strategy tables, as they would not depend on the set of lost functions but only on their number (in the previously presented schemes the state of the look-up tables depended on the first factor).

However, the possibility of certification of such reconfiguration algorithms may be questionable, as functions of identical criticality would be required to reconfigure to one another. If such solutions are not acceptable from the safety and certification point of view, then communication based methods should be chosen or the capacity for module recovery should not be allowed in schemes based on dynamically updated strategy data.

Commentary:
It might be acceptable for some systems to define preferences for some functions over others of the same criticality (e.g. despite the fact that the failure conditions related to the loss of functions F1 and F2 are minor, the crew workload will be higher if function F1 is lost, and therefore it should be sustained at the expense of the loss of function F2). However, in the general case such relationships between functions would depend on the flight phase and/or the state of the cabinet and thus it might be difficult to identify. This would require more effort and would add additional complexity to the design and generation of the reconfiguration data.

The recovery problems discussed above exist also in reconfiguration methods employing module based strategy tables, and are again related to the possibility of discontinuities occurring in the table columns. As in the function based schemes, if avionics functions of same criticality were allowed to reconfigure to one another, the problem could be easily eradicated. Again this may not be acceptable, and alternative ways of dealing with module recovery may have to be employed.

In module based strategy tables each core LRM always refers to the same matrix row to determine its actions on a function loss. In such a case, it is conceivable that the recovering module could operate based on its own row only, and without any knowledge of other modules reconfiguration policies.

Commentary:
As in the function based method, the module could set backup levels for each function it has got the relevant software for to "0", and then send an appropriate message on the backplane bus to allow other modules to update their data. However, unlike in the previous method, the module would not attempt to recreate data for other modules, but would simply assume that all other entries are equal to "-1" (not in backup).

However, some other problems may occur, as in order to be able to update the strategy table on following failures, the recovered core LRM must be able to determine whether the lost function has got a backup or not (see section 6.3.2.1 for discussion on required update procedure). On the other hand, if the module is not in backup for the function (lack of software), its loss will not affect the module operation and can be ignored. If the module was in backup, it will be in the immediate backup (all entries were reset to zero), and thus it will be able to identify the reconfiguring module - self. Should another module fail and recover, the module will be notified by an appropriate message, and it will be able to update its strategy table accordingly. The module will therefore have all the knowledge about backup levels lower than its own, and no information about higher levels will be required.

Although the above solution is theoretically plausible, it exhibits some inherent safety risks related to the use of incomplete reconfiguration data (even though the data remains consistent with respect to the reconfiguration algorithm), and could thus prove difficult to certificate. If again the communication based methods are not acceptable or feasible and the approach allowing functions of the same criticality to be reconfigured is not desirable, the recovery in dynamic module based reconfiguration schemes should be avoided.

#### 6.3.2.3. Discussion and comparison of dynamic data based reconfiguration schemes

Both of the above presented approaches lead to shortest possible reconfiguration delays, i.e.

- if there exist backup modules for a function, one of them will always be in an immediate backup mode,
- all avionics functions are stored locally in the core LRM non-volatile memory to avoid delays related to software downloading.

However, to achieve their time objectives they need to dynamically update their reconfiguration strategy data, that introduces concerns about data consistency and possible data obsolescence (the latter problem refers particularly to reconfiguration methods allowing module recovery). Such behaviour will complicate the failure analysis and certification procedures, as sources and effects of possible loss of data consistency need to be identified and their consequences assessed.

In the case of function based look-up tables, the reconfiguration routine and related updates are relatively simple and affect the whole rows and columns of the strategy matrix. The updates of the data are highly deterministic, and the state of the look-up table depends merely on the set of the lost functions. In the case of module based approaches, additional functionality is required from the data bus monitoring routines to allow preservation of data consistency. The modules must detect not only function losses in order to be able to update the table in the event of reconfiguration, but also need to detect module failures to note the loss of backup for particular avionics functions. Moreover, whilst in the function based schemes the data updates relate uniformly to the whole rows and columns of the strategy table, in this approach partial row (selected columns only) updates are required, as the reconfiguring module stops providing backup only for these of the functions for which it has got the application software.

Moreover, similarly to the static module based reconfiguration methods, long reconfiguration chains may follow single module failures (see the sequence of events described in section 6.3.1.3 for an example). As this can be perceived as a generic deficiency of module based methods, the discussion from section 6.3.1.3 holds for dynamic module based algorithms as well as the static methods. Also, unlike the function based approach that exhibits a highly desirable property of deterministic functional system degradation, the behaviour of the module based schemes (both static and dynamic) differ depending on the current state of the cabinet. For instance, in a module based system with a redundant core LRM reconfigured to perform an avionics function, further reconfiguration actions will be different than the those of the same system with the redundant module failed[59].

Although the dynamic data based reconfiguration methods bring the benefit of short reconfiguration delays, they introduce considerable difficulties when module recovery is required. Schemes implementing recovery will require either explicit transfer of strategy tables from operating core LRMs to the recovering one, or methods for recreating the data. In the first approach the autonomy of each core LRM can be questioned (its future reconfiguration actions will depend on information from other processing modules), and an additional communication overhead can be expected[60]. In the latter solution various other problems may arise, such as the need for functions of identical criticality being reconfigured or the possibility of core LRMs operating with incomplete reconfiguration data. Both issues have to be considered as a serious obstacle when certification of dynamic data methods with capacity for module recovery is being considered. Also, in either approach the complexity of the reconfiguration software will increase significantly, which may prohibit the use of formal proof methods for their verification.

Both methods (module and function orientated) introduce certain safety hazard with their capacity for module recovery, as the recovered module is set-up to provide an immediate backup for a number of avionics functions. That, in turn, raises questions about the confidence that the module will not fail again (a generic problem when a core LRM is allowed to recover from a transient failure), but also imposes a

[59] Note that both cabinets will perform exactly the same avionics functions.

[60] Based on the size of the reconfiguration strategy data and the number of core LRMs per cabinet this can be expected to be relatively insignificant.

certain safety risk as the "weak" module is likely to reconfigure first[61] in the event of a subsequent loss of a critical function.

It has to be concluded, that although dynamic data based reconfiguration schemes operate with shortest possible delays, they are generally much more complex than static data based methods, particularly if they allow for module recovery. Therefore, their use may be recommended for strict real-time systems such as RIMA, but only in the case of very short allowed function absence times where static methods are not able to implement reconfiguration within the time constraints.

## 6.4. Conclusions

In the previous sections various reconfiguration methods based both on static and dynamic strategy data were presented and discussed. The choice between the two approaches comes as a trade off between reducing the method complexity and shortening the delays.

Preliminary tests conducted on a software model of RIMA suggest that schemes employing static reconfiguration data can be designed and implemented in a less complex manner than the dynamic methods. This follows the fact that static strategy table based schemes do not update their strategy data (such updates become particularly complex in module orientated schemes), and thus do not require some of the functionality that is essential in dynamic data based schemes. Also, the capacity for module recovery can be provided without any additional effort for the static schemes, but it introduces considerable difficulties in dynamic methods as additional code is required to maintain the coherency of the evolving strategy tables.

Furthermore, the core of the algorithm – the reconfiguration strategy data can be generated for a given system during the integration phase, and its correctness can be assessed before the system comes on line. Once verified, the data remains valid for the whole life of a RIMA system, i.e. it will not require any modifications unless the cabinet configuration changes. Moreover, in this approach the introduction of the capacity for module recovery does not lead to a significant increase of the algorithm complexity.

[61] This will obviously depend on the software availability.

Therefore, due to their simplicity, certification of reconfiguration schemes employing static strategy tables should allow for the use of formal proof methods.

However, in certain cases where only very short reconfiguration delays are allowed, static data based methods may not be applicable.

> Commentary:
> If the time allowed for the backup module to undertake the reconfiguration decision is $T$, for each failure detection $K$ missing messages are required (see Chapter 8 for discussion on failure detection mechanisms), and up to $N$ backup levels must be provided, the time interval $\Delta t$ between consecutive messages may become very small ($\Delta t=T/(N*K)$). Particularly for critical functions (large $N$) with very short allowed absence time (small $T$), this could lead to unfeasible bus access schedules for synchronous data buses, or to a denied bus access in the case of asynchronous ones.

In such situations the dynamic data based reconfiguration methods need to be considered. In this approach the time interval between messages does not depend on the required number of backup levels ($\Delta t=T/K$), as there always exists an immediate backup core LRMs for each sufficiently critical avionics function. This is achieved by modifying the entries of strategy table to reflect the current state of the cabinet (for example a loss of a processing module), but as discussed earlier it requires additional implementation effort increasing the overall complexity of the code. Thus, the choice between static and dynamic strategy tables can be understood as a trade-off between low software complexity (static methods) and short reaction times in an event of a failure (dynamic methods).

However, dynamic solutions should be avoided in systems for which static methods are applicable. As discussed above, dynamic data based reconfiguration algorithms are more complex than their static equivalents, particularly if the capacity for module recovery is required. Also, as the reconfiguration strategy table evolves throughout the system life and its invalidity may expose the aircraft to a safety hazard, the update algorithms will require rigid verification procedures (possibly a formal proof), to ensure the data consistency in any combination of failures and recoveries[62]. Moreover, as the strategy data could become incoherent due to implementation errors, i.e. although the update mechanism is proven

[62] If module recovery is to be allowed, the cost related to highly complex proof and verification procedures may outweigh all the gained benefits.

correct, the actual software may contain errors either due to a programming mistake or compiler problem. It is likely that dynamic data based reconfiguration schemes will require some consistency maintenance routines, that will further increase method complexity. Finally, when dynamic module based reconfiguration tables are being considered, the method itself will require additional functionality to allow for monitoring of core LRMs and functions they are performing, that will complicate the certification process even more.

The following Table 6.2 summarises the properties of particular reconfiguration schemes.

| Method | Delays | Complexity |
|---|---|---|
| **Dynamic data method:** module orientated strategy tables | Short | Complex table updates (including partial row updates)<br>Requires additional monitoring of core LRMs for reconfiguration,<br>Recovery greatly increases algorithm complexity and may lead to undesirable effects (reconfiguration of functions of same criticality)<br>Requires dedicated recovery messages or recovery related communication |
| **Dynamic data method:** function orientated strategy tables | Short | Requires relatively simple row and column updates,<br>Recovery greatly increases algorithm complexity and may lead to undesirable effects (reconfiguration of functions of same criticality)<br>Requires dedicated recovery messages or recovery related communication |
| **Static data method:** module orientated strategy tables | Depending on the backup level | Very simple (no data updates) |
| **Static data method:** function orientated strategy tables | Depending on the backup level | Very simple (no data updates)<br>Exhibits a highly desirable property of deterministic functional system degradation (predictable system evolution) |

**Table 6.2. Reconfiguration delays and complexity of proposed reconfiguration schemes.**

Due to their simplicity, static data based methods are expected to constitute the preferred choice for implementation in RIMA. However, in systems where they are unable to meet the strict time

requirements, dynamic data methods may have to be employed. Also, if the choice between the module and function orientated reconfiguration strategy tables is to be made, the use of function based solutions is strongly advised, as they allow reconfiguration chains to be limited to a single reconfiguration, can be generally implemented in a rather simple manner, and finally offer a very desirable property of functional deterministic system degradation. Therefore, unless there are other physical limitations (e.g. insufficient amount of non-volatile memory per core LRM), the static function orientated reconfiguration methods are strongly recommended.

# Chapter 7. Formal Specification of a Reconfiguration Scheme

## 7.1. Introduction

Reconfigurable Integrated Modular Avionics systems (RIMA) employ dynamic reconfiguration schemes in order to provide fault tolerance with reduced processing module redundancy (see Chapter 4). The scheme therefore becomes safety-critical, and formalising its description allows one to gain confidence that the scheme will meet all the system requirements. The formal description of the method constitutes also a good basis for further proofs, reasoning about the scheme properties, and for the implementation of the scheme.

This chapter provides an informal description of a reconfiguration scheme in a natural language (English), as well as its formal description with the use of the Vienna Development Method (VDM) specification language [43],[44], [45]. The scheme presented in this chapter is based on the principles discussed in Chapter 5 and Chapter 6, and it utilises a static function based strategy look-up table to govern reconfiguration. The choice follows the argument presented in [46] and commonly shared in the computer systems society that the simplicity is the key issue where reliability of a system is being considered.[63]

The remaining part of the report is organised as follows. Section 7.2 gives an informal but detailed description of the scheme, whilst section 7.3 provides its formal VDM specification.

## 7.2. Natural language description of the reconfiguration scheme

This section discusses the static strategy data and function orientated reconfiguration scheme based on the principles identified in the previous chapter. Processing modules employ a look-up table to determine whether they should or should not reconfigure on detection of another module failure. The strategy look-

[63] The discussion from Chapter 6 and the results of preliminary tests of various schemes, with the aid of a software model of RIMA, indicate that static function based schemes can be implemented in an extremely simple manner.

up table and the general design of the scheme have been discussed in chapter 6. At this the design blocks and their particular routines will be presented and discussed in a greater detail.

## 7.2.1. Description of the Initialisation and Recovery Block

The main block is responsible for initialisation of the reconfiguration scheme during the cabinet power-up sequence or on module recovery. Having initialised the scheme, the **main** routine of the block is further responsible for distributing control to other functional blocks.

### 7.2.1.1. Function main

Function **main** initiates the assignment of avionics functions to core LRMs. It is also used to select an appropriate function on module recovery, thus making recovery transparent to the algorithm. The function initialises the reconfiguration data resetting the responses from all avionics functions. Subsequently, it monitors the backplane bus for the time interval of (N+1)×RECOVERY-DELAY, where N denotes module ID[64], in order to determine functions that are not being performed in the cabinet. After all initial monitoring is completed, the module selects from the lost functions the one with the lowest ID and starts performing it. Such a mechanism ensures not only that the most critical lost function will be selected, but also determines the order functions of identical criticality are selected.

> Commentary:
> Although the above algorithm had been developed independently, it coreseponds closely to the election protocol as seen in [40]. Note that the system implementing such a protocol must guarantee that a delay related to sending and receiving a message will not be longer than a given constant (in this example RECOVERY-DELAY).

Having dealt with the initialisation issues, the **main** function is then used to distribute control to other functional blocks (Data Bus Monitoring Block and Reconfiguration Block).

[64] The ID is based on module position in the rack and can be hardcoded in the module non-volatile memory. Note that the higher the ID, the lower criticality function will be selected on system initialisation.

## 7.2.2. Description of the Data Bus Monitoring Block

The Data Bus Monitoring block keeps track of responses from all the avionics functions recording them in appropriate data structures.

### 7.2.2.1. Function get-message

Function **get-message** attempts to read a message from the backplane bus buffer. All messages except those dedicated to reconfiguration are ignored. Messages conferring states of avionics functions are recorded and stamped with the current time by the function **save-state**. Messages from particular avionics functions (including state messages) are recorded as the latest activity from the function by the **record-response** routine.

### 7.2.2.2. Function record-response

This function is called with the ID of the function for which the response is to be recorded as a parameter. The current time value is used to stamp the response in the response time array for the appropriate avionics functions.

### 7.2.2.3. Function save-state

The function saves the state data for a given avionics functions function in the designated array. In order to eliminate the necessity for the function to understand the state for each avionics application, it deals with the state as a raw byte-by-byte data. This function saves the contents of the state message in the memory and stamps it with the current time. For each function requiring state a STATE-AGE value is defined during system integration, and thus should the state be older than its maximum allowed age, a default state will be used on function initialisation.

## 7.2.3. Description of the Reconfiguration Block

### 7.2.3.1. Function check-reconfiguration

This function is called with the module current function ID as a parameter. It returns true if the module should reconfigure based on the current reconfiguration data (responses and strategy), i.e. the reconfiguration conditions are satisfied with respect to the lost function. Otherwise it returns false.

#### 7.2.3.2. Function detect-failure

This function is called with the module current function ID as a parameter, and it can only be called if check-reconfiguration returned true for this function ID. The routine returns the ID of the function to which the module should reconfigure.

#### 7.2.3.3. Function reconfigure

This function undertakes the reconfiguration decisions based on the strategy data. It checks whether the module should reconfigure, calling the **check-reconfiguration** routine with the ID of the avionics function currently being performed as a parameter. Should **check-reconfiguration** return true, the **detect-failure** is called in order to obtain the ID of the new function. This function then terminates the old avionics function performed by the core LRM, and restores execution of the new application based on its most recently saved state. The **reconfigure** routine returns SUCCESS if reconfiguration was required and was successful, NO_RECONFIGURATION if reconfiguration was not required, and FAILURE if reconfiguration was required but was unsuccessful. This can be related, for example, to problems with the application executive (APEX), the operating system (OS), the lack of system resources, corrupted application software or invalid state.

The information about the state of the new function should be retrieved from the state array, taking into account the age of the most recently saved state, i.e. if the recorded state is older than allowed, a default initial state for the function should be used. Applications that do not require the state should be treated as applications with default state.

### 7.2.4. Description of the data items required to implement the complete scheme

To avoid any bias towards a particular implementation, all data items discussed at this point include only structures required by the functionality of the reconfiguration scheme, and they do not include variables and auxiliary data required to implement particular functions or algorithms.

#### 7.2.4.1. Description of the strategy data

The reconfiguration strategy look-up table governs the order of reconfiguration and it determines the module behaviour on detected failures. Its use and creation principles have been discussed in detail in

Chapter 6. The strategy table used in this approach is static, i.e. it is not to be updated or changed by core LRMs at any point of system operation, and it is function orientated.

#### 7.2.4.2. Description of the response time array

The response array holds information about the latest responses from particular functions. It can be visualised as a single dimensional array, whose entries correspond to particular avionics functions. On receiving a message from an avionics function, the value of the corresponding entry will be updated with the current value of the internal clock (timer).

> Commentary:
> It is important from the implementation point of view, that the maximum value of the clock is big enough to support operation of the algorithm for a sufficiently long time. It is also essential that the accuracy of the timer allows for calculation of time differences at least in milliseconds (time intervals between consecutive messages can be as short as 5 - 10 ms). Timer/clock services are expected to be provided by the OS/APEX implementation, and as such are listed in section 7.2.5.

#### 7.2.4.3. Description of the state array

The state array will contain all the required information about the state of all avionics functions in the cabinet. This will include:

- the size of the state and its contents (for example an address to memory where it is stored or the name of the appropriate variable)
- the time when the last state information has been saved for a given function
- the STATE-AGE value (if the saved state is older than the allowed age, a default state will be used instead).

If the state size is equal to zero, the application is said to be stateless. It is expected that no information about default initial state will have to be stored, as the avionics functions executed without state information will automatically fall back to their default initial state.

### 7.2.5. Required system services

In order to implement the scheme the following OS/APEX functionality needs to be provided:

- incremental timer, with capacity unlimited in terms of a single system run
- services for reading messages from the backplane bus buffer, including extraction of data and information about their sources

- APEX services for dynamic process/function management.

These are expected to be reconfiguration algorithm independent.

## 7.3. VDM-SL specification of scheme data and functions

The Vienna Development Methods Specification Language (VDM-SL) has been chosen for the purpose of formalising the reconfiguration scheme. The choice follows the fact that VDM is one of the most mature and standardised formal methods in software engineering at the present time ([43], [44], [45]), and various tools can be used to verify the syntax and some semantics of the specification. In this project the IFAD VDM-SL Toolbox [47], [48] and the Adelard Spec Box [49] had been used to check the syntax correctness of the specification.

The remainder of this section provides the formal specification of the data and functions described in section 7.2, as well as some additional functions and data structures necessary to make the specification complete.

The additional functions can be treated as auxiliary, and their use follows the desire of avoiding repeating the same statements in many different functions and operations (e.g. function state-message(...) identifies the given message as a valid state of an avionics function, and it is used in operations MAIN and SAVE-STATE). Other functions such as has-all-entries(...) or do-states() are included in the specification in order to properly define the reconfiguration data, thus allowing for automatic verification of the specification by VDM-SL orientated tools. Note that in the actual implementation of the scheme, the strategy data will not be created at the system power-up (functions do-strategy() and get-level() will not be required), but it will have been defined during system integration. Note also, that the operation MAIN does not explicitly reset all the state variables, as in VDM the state is initialised automatically by the states initialisation function (init-RIMA in this case).

In the following specification the state contains some variables which do not strictly belong to the scheme itself such as the buffer, currentTime and systemError. These data structures relate to OS/APEX services (see section 7.2.5), and are included in the specification in order to be able to portrait the interaction of

the reconfiguration software and its environment. Constants such as FAIL-DELAY, RECOVERY-DELAY, maxID, startTime and startID represent only some possible values for those parameters, and in the actual implementation these values will depend on the system being developed, its size, time constraints and the particular module.

In the following section all comments not being a part of the specification will be enclosed in a bounding box similarly to this note.

### 7.3.1. VDM-SL specification of the scheme

```
module RIMAMOD
     exports all
definitions
```

The values below should be understood as follows:

- maxID - A natural number representing the highest ID of a core LRM or an avionics function (including the redundant ones). As function/module IDs start from zero, maxID + 1 is equal to the number of processing modules in the cabinet.
- FAIL-DELAY - The time interval for which there must be no activity from an avionics function to be able to announce it as lost. Although in a general case FAIL-DELAY may vary for each function, in this specification it is identical for all functions and equal to 50 ms purely for the simplicity reasons. In order to allow varying fail delays in the system to be implemented an appropriate data structure (delays : DelayArray) is provided, whose entries represent FAIL-DELAY for particular avionics functions.

```
values

maxID : ℕ = 9;
FAIL-DELAY : Time = 50;
```

The data types below should be understood as follows:

- ID - a natural number between zero and maxID inclusive, representing the ID of a core LRM or an avionics function
- Level - an integer number between -1 and (maxID-1) inclusive, representing a backup level for an avionics functions as used in the strategy table
- Time - a natural number representing system time in the same time units as FAIL-DELAY (here milliseconds)
- Pair - a pair of IDs used as an index to a two-dimensional strategy look-up table
- DelayArray - a single-dimensional array with indexes ranging from zero to maxID, and values representing fail delays for avionics functions, where array index corresponds to function ID
- ResponseArray - a single-dimensional array indexed as above, but with values representing time stamps of the most recent messages received from avionics functions
- StateInfo - time stamped state of an avionics function
- StateArray - a single-dimensional array similar to the DelayArray, but with values representing the most recently saved states of avionics function, indexing as for the DelayArray
- Error - enumerated, simplified representation of possible OS/APEX reconfiguration related error values
- Status – enumerated, status of execution of the reconfiguration routine
- MsgType – enumerated, possible types for messages received from the backplane bus; note that only messages of type STATE_MSG (message conveying a function state) and RESPONSE_MSG (message constituting a valid function activity) are relevant to reconfiguration, and all other messages are marked as ANY_MSG
- Message - a record type depicting the header and the contents of a message; note that in order to be useful to the reconfiguration algorithm, each message must have its source set to the ID of the function it had originated from
- Buffer - a data type representing a queue of messages and corresponding to the backplane bus buffer provided by OS/APEX in RIMA
- Strategy - strategy look-up table as explained in Chapter 6.

```
types

ID = ℕ

inv id≜

        id ∈ {0,...,maxID};
```

```
Level = ℤ

inv level ≜
        level ∈ {-1,...,maxID-1};


Time = ℕ;


Pair ::    f1ID : ID
           f2ID : ID;


DelayArray = ID ──m──→ Time

inv delayarray ≜
        dom delayarray = {0,...,maxID};


ResponseArray= ID ──m──→ Time

inv responsearray ≜
        dom responsearray = {0,...,maxID};


StateInfo :: time-stamp : Time
             contents : char*;


StateArray = ID ──m──→ StateInfo

inv statearray ≜
        dom statearray = {0,...,maxID};


Error    = <NO_ERROR> | <ERROR>;


Status     = <NO_RECONFIGURATION> | <SUCCESS> | <FAILURE>;


MsgType    = <STATE_MSG> | <RESPONSE_MSG> | <ANY_MSG>;


Message :: type : MsgType
           source : ID
           contents : char*;


Buffer = Message*;
```

```
Strategy = Pair ──m──→ Level
inv s ≜
        has-all-entries(s) ∧
        has-continuous-columns(s) ∧
        has-sufficient-backups(s)
```

The values below should be understood as follows:

- startTime - a natural number used to initialise the system clock during module power-up
- startID - the hardcoded ID of the core LRM used during scheme initialisation; each ID is a cabinet unique natural numbers associated with the processing unit; IDs cover the whole range from zero to the number of modules in the cabinet less one.

```
values

startTime :   Time = 0;
startID :     ID = 0
```

The following definitions represent the state of the specification. Particular fields should be understood as follows:

- strategy - the strategy look-up table as explained in Chapter 6
- states - an array holding the most recently received states of all avionics functions
- responses - an array holding times of the most recently observed activity from all avionics functions
- delays - an array holding the values of FAIL-DELAY for all avionics functions
- buffer - the message buffer for the backplane bus, a part of the system services
- currentID - the ID of the avionics function currently performed by the core LRM
- currentTime - the current value of the system clock, a part of the system services
- systemError - the variable used to represent the state of the system, it will be set by OS/APEX to an error ID (here represented as ERROR) to indicate OS/APEX software/hardware problems

```
state RIMA of
        strategy    : Strategy
        states      : StateArray
        responses   : ResponseArray
        delays      : DelayArray
        buffer      : Buffer
        currentID   : ID
        currentTime   : Time
        systemError   : Error
inv rima≜
        inv-Strategy(rima.strategy)
init rima ≜ rima = mk-RIMA(do-strategy(),do-states(startTime),
        do-responses(startTime),do-delays(FAIL-DELAY),[],startID,
        startTime,<NO_ERROR>)
end
```

A brief description of the number of the following auxiliary functions is given at this point. Other functions are described on per case basis.

- entry - auxiliary VDM function to provide an easy an readable way of creating two-dimensional indexes to the strategy look-up table
- in-critical - function returns true if the failure conditions following the loss of a function of the ID given as the argument are critical; as the results of this and the following four in-XXX functions are clearly dependent on the cabinet being integrated, the following assumptions about functions criticality in terms of failure conditions have been made in this specification: function 0 - critical, functions 1, 2, 3 - hazardous, functions 4, 5 - major, functions 6, 7 - minor, functions 8, 9 - redundant
- in-hazardous - function returns true if the failure conditions following the loss of a function of the ID given as the argument are hazardous
- in-major - function returns true if the failure conditions following the loss of a function of the ID given as the argument are major
- in-minor - function returns true if the failure conditions following the loss of a function of the ID given as the argument are minor
- in-redundant - function returns true if the ID given as the argument denotes a redundant pseudo-function
- criticality - function returns the criticality of the avionics function of the ID given as the argument in terms of the severity of the failure conditions following the loss of the function; the returned value should be understood as follows: 0 - critical, 1 - hazardous, 2 - major, 3 - minor, 4 – redundant (no safety hazard)
- required-backup - function returns the number of backups required for the function of a given ID necessary to meet the safety requirements

```
functions

entry(f1:ID, f2:ID) r:Pair
post r = mk-Pair(f1,f2);


in-critical(fID:ID) r:B
post r = (fID ∈ {0});


in-hazardous(fID:ID) r:B
post r = (fID ∈ {1,2,3});


in-major(fID:ID) r:B
post r = (fID ∈ {4,5});


in-minor(fID:ID) r:B
post r = (fID ∈ {6,7});


in-redundant(fID:ID) r:B
post r = (fID ∈ {8,9});


criticality(fID:ID) r:ℕ
post r = if in-critical(fID)
         then 4
         else if in-hazardous(fID)
              then 3
              else if in-major(fID)
                   then 2
                   else if in-minor(fID)
                        then 1
                        else 0;
```

```
required-backup(fID : ID) r : ℕ
     post r = if in-critical(fID)
        then 4
        else if in-hazardous(fID)
             then 4
             else if in-major(fID)
                  then 3
                  else if in-minor(fID)
                       then 2
                       else 0;
```

The functions below - has-all-entries(...), has-continuous-columns(...) and has-sufficient-backups(...) are used to implicitly define the properties of the strategy look-up tables. In order to avoid bias towards implementation, the functions do not define an algorithm for creation of the strategy table, but define the strict conditions which have to be satisfied by a valid strategy table.

```
has-all-entries(table: Pair ──m──→ Level) r: B
     post r = (∀ i,j : ID • entry(i,j) ∈ dom table);
```

```
has-continuous-columns(table : Pair ──m──→ Level) r:B
     post r =(∀ i,j : ID •
          (criticality(i) < criticality(j) ∧ table(entry(i,j))=(maxID - i)) ∨
          (criticality(i) ≥ criticality(j) ∧ table(entry(i,j))=-1));
```

```
has-sufficient-backups(table : Pair ──m──→ Level) r:B
     post r = (∀ i:ID • ∃ j:ID • table(entry(j,i))=required-backup(i)-1);
```

The following two functions - get-level(...) and do-strategy(...) give an algorithmic definition of the strategy look-up table, allowing the VDM oriented tools for easy initialisation of and verification of the state.

```
get-level(i:ID,j:ID) r:Level
     post if (criticality(i) < criticality(j))
        then r = (maxID - i)
        else r = -1;
```

```
do-strategy() r:Strategy
      post r = {entry(i,j) ↦ get-level(i,j) | i,j ∈ {0,...,maxID}};
```

Functions do-states(...), do-responses(...) and do-delays(...) are used during initialisation of the appropriate components of the state (arrays states, responses and delays respectively).

```
do-states(t : Time) r:StateArray
      post r = { i ↦ mk-StateInfo(t,[]) | i ∈ {0,...,maxID}};

do-responses(t : Time) r:ResponseArray
      post r = { i ↦ t | i ∈ {0,...,maxID}};

do-delays(d : Time) r:DelayArray
      post r = { i ↦ d | i ∈ {0,...,maxID}};
```

The four functions below should be understood as follows:

- state-message - function returns true if the message given as an argument is a valid state message
- response-message - function returns true if the message given as an argument corresponds to a valid activity of an avionics function
- lost-delays - function returns the number of FAIL-DELAYs for which a function of the ID given as an argument has been lost; function uses the following state components passed to it as arguments: responses, delays, currentTime
- in-lost - function returns true if the avionics function of the ID given as an argument has been lost for at least one FAIL-DELAY defined for this function; the function uses the following state components passed to it as arguments: responses, delays, currentTime.

```
state-message(msg : Message) r:B
      post r = (msg.type = <STATE_MSG> ∧ msg.contents ≠ [] ∧
         msg.source ∈ {0,...,maxID});

response-message(msg : Message) r:B
      post r = (state-message(msg) ∨ msg.type = <RESPONSE_MSG> ∧
         msg.source ∈ {0,...,maxID});
```

```
lost-delays: ID × ResponseArray × DelayArray × Time → ℤ

lost-delays(fID,responses,delays,time) ≙
        (time - responses(fID)) div delays(fID);


in-lost: ID × ResponseArray × DelayArray × Time → B

in-lost(fID,responses,delays,time) ≙
        (lost-delays(fID,responses,delays,time) ≥ 1);
```

The following functions - **check-reconfiguration** and **detect-failure** are based on the natural language description as given in the previous section. Note that some of the conditions from both functions evaluate to the same logical statement (e.g. lines 40.4 and 40.5), but has been included to reduce the risk of the scheme failure (e.g. the contents of the strategy table is susceptible to modification by natural phenomena) and to improve the clarity of the specification.

```
check-reconfiguration(fID:ID,strategy:Strategy,responses:ResponseArray,
        delays:DelayArray,time:Time) r:B
     post r = ∃ f : ID •
          in-lost(f,responses,delays,time) ∧
          (lost-delays(f,responses,delays,time)-1) ≥strategy(entry(fID,f))∧
          strategy(entry(fID,f)) ≠ -1 ∧
          criticality(fID) < criticality(f)


detect-failure(fID:ID,strategy:Strategy,responses:ResponseArray,
        delays:DelayArray, time:Time) r:ID
   pre check-reconfiguration(fID,strategy,responses,delays,time)
     post in-lost(r,responses,delays,time) ∧
          ((lost-delays(r,responses,delays,time)-1)≥ strategy(entry(fID,r))) ∧
          strategy(entry(fID,r)) ≠ -1 ∧
          criticality(fID) < criticality(r) ∧
          ∀ f : ID •
             in-lost(f,responses,delays,time) ∧ strategy(entry(fID,f))≠-1) ∧
             lost-delays(f,responses,delays,time)-1 ≥ strategy(entry(fID,f)) ⇒
             (criticality(r) ≥ criticality(f));
```

The operations below should be understood as follows:

- GET-MESSAGE - extracts the first message from the backplane bus buffer queue
- SAVE-STATE - saves into the states array the state information contained in the message given as an argument; the operation also time stamps the state information with the current value of the system clock
- RECORD-RESPONSE - operation records the activity from the function from which the message given as an argument originated, updating the corresponding entry in the array 'responses' with the current value of the system clock.

```
operations

GET-MESSAGE() r : Message
      ext wr buffer : Buffer
      post if ~buffer~ ≠ []
            then r = hd ~buffer~ ∧ buffer = tl ~buffer~
            else r = mk-Message(<ANY_MSG>,0,[]) ∧ buffer = ~buffer~;


SAVE-STATE(msg : Message)
      ext wr states : StateArray
         rd currentTime : Time
      post if state-message(msg)
            then states=
               ~states~ † {msg.source ↦ mk-StateInfo(currentTime,msg.contents)}
            else states = ~states~;


RECORD-RESPONSE(msg : Message)
      ext wr responses : ResponseArray
         rd currentTime : Time
      post if response-message(msg)
            then responses = ~responses~ † {msg.source ↦ currentTime}
            else responses = ~responses~;
```

The operation RECONFIGURE performs the actual reconfiguration. If the reconfiguration conditions are satisfied (check-reconfiguration(...) returns true), the operation selects the new avionics function to be performed with a call to the function detect-failure(...). The current function ID is then set to the new value, and if no system errors have occurred the operation returns SUCCESS as its result, otherwise it returns FAILURE. In the case where the reconfiguration conditions are not satisfied the operation will return NO_RECONFIGURATION as its resultant status. Note that the notion of function state is not handled explicitly, as the implementation details of execution of the new application are not discussed at this point.

```
RECONFIGURE() r:Status
      ext wr currentID : ID
         rd strategy : Strategy
         rd responses:ResponseArray
         rd delays:DelayArray
         rd currentTime:Time
         rd systemError:Error
      post if
      check-reconfiguration(currentID,strategy,responses,delays,currentTime)
         then let newID =
            detect-failure(currentID,strategy,responses,delays,currentTime) in
               if systemError = <NO_ERROR>
               then currentID = newID ∧ r = <SUCCESS>
               else currentID = currentID ∧ r = <FAILURE>
      else r = <NO_RECONFIGURATION>;
```

The following definitions of the function **detect-failure2**(...) and the operation **RECONFIGURE2**() provides an alternative specification for the corresponding definitions of **detect-failure**(...) and **RECONFIGURE**(). The alternative specification avoids pre-conditions, i.e. the function can be called and its results are defined in any circumstances, and as such can be considered as possibly more robust[65].

[65] In [50] the author suggests that pre-conditions should be avoided in specifications of critical systems, as the behaviour of a function called in a state where the function pre-condition does not hold is undefined.

```
functions

detect-failure2(fID:ID,strategy:Strategy,responses:ResponseArray,
          delays:DelayArray,time:Time) r : ID*
     post if check-reconfiguration(fID,strategy,responses,delays,time)
        then let id:ID be st (in-lost(id,responses,delays,time) ∧
          ((lost-delays(id,responses,delays,time)-1)≥strategy(entry(fID,id)))∧
          strategy(entry(fID,id)) ≠ -1 ∧
          criticality(fID) < criticality(id) ∧
          ∀ f : ID •
          (in-lost(f,responses,delays,time) ∧ strategy(entry(fID,f))≠-1) ∧
          lost-delays(f,responses,delays,time)-1 ≥ strategy(entry(fID,f))⇒
             (criticality(id) ≥ criticality(f))) in
               r = [id]
        else r = []

operations

RECONFIGURE2() r:Status
     ext wr currentID : ID
        rd strategy : Strategy
        rd responses:ResponseArray
        rd delays:DelayArray
        rd currentTime:Time
        rd systemError:Error
post if
check-reconfiguration(currentID,strategy,responses,delays,currentTime)
   then let newID =
detect-failure2(currentID,strategy,responses,delays,currentTime) in
      if newID ≠ [] ∧ systemError = <NO_ERROR>
         then currentID = hd(newID) ∧ r = <SUCCESS>
         else currentID = ~~currentID~~ ∧ r = <FAILURE>
      else r = <NO_RECONFIGURATION>
```

The remainder of the specification refers to the **main** function of the reconfiguration scheme. The value of **RECOVERY-DELAY** describes the time interval used during initialisation on system start-up or module recovery, as explained in section 7.2.1.1. Function **get-id**(...) returns the lowest ID from the set of all functions not being performed in the cabinet. Operation **MAIN** behaves as explained in section 7.2.1.

```
values

RECOVERY-DELAY : Time = 100


functions

get-id(responses : ResponseArray, delays : DelayArray, time : Time) r:ID
     post in-lost(r, responses, delays, time) ∧ ∀ f:ID •
        (in-lost(f, responses, delays, time) ∧ r ≠ f) ⇒ (r < f)


operations

MAIN : () ⇒ ()
     MAIN() ≜
        (dcl status : Status := <NO_RECONFIGURATION>;
         dcl msg : Message := mk-Message(<ANY_MSG>,0,[]);
           while currentTime ≤ startTime + (currentID+1) × RECOVERY-DELAY do
              (msg := GET-MESSAGE();
              (if state-message(msg) then SAVE-STATE(msg)
              else RECORD-RESPONSE(msg)));
           currentID := get-id(responses, delays, currentTime);
           while true do
              (msg := GET-MESSAGE();
              (if state-message(msg) then SAVE-STATE(msg)
              else RECORD-RESPONSE(msg));
              status := RECONFIGURE();
              (if (status = <FAILURE>) then exit
              else skip )))

end RIMAMOD
```

# Chapter 8. Formal Discussion of the Reconfiguration Scheme

## 8.1. Introduction

In this section various properties and aspects of the reconfiguration scheme specified in the previous chapter are discussed. They include safety-critical requirements identified for the reconfiguration scheme and the analysis of key parts of the scheme with respect to their reliability, limitations and timing constraints.

The notation used whilst referring to functions and operations from the specification given in Chapter 7 will describe them by their number as seen in the specification (e.g. s.45). Particular statements will be referred to as s.45.2, where the last number denotes the line of the definition of a function or operation. In the cases where a function is referred to by its name, only the relevant arguments will be stated explicitly, and other positions in the function argument list will be denoted by "...".

Following the VDM style, statements referring to functions returning Boolean results (e.g. *fn(a1,a2,...)=TRUE*) will often be replaced by the call to the function *fn(a1,a2,....)*, what yields the same logical meaning.

## 8.2. Properties of the Scheme

Various properties of the previously defined scheme will be shown in this section. As the state variable systemError represents the state of the physical system (i.e. corresponds to any specification independent hardware or software faults), its behaviour cannot be predicted. Therefore, in order to be able to reason about the reconfiguration algorithm, it will be assumed to always read NO_ERROR.

> Commentary:
> systemError corresponds only to those system faults that directly interfere with the reconfiguration software (in this case OS/APEX failure to spawn a new function process). Therefore, it is safe to reason about the behaviour of the reconfiguration scheme when the error flag is not set, as in the case of a serious system error, its behaviour will be of little interest (the system will be expected to shutdown in such a situation). However, a failure of the backplane bus to deliver a message is not

considered as a system fault, thus the scheme behaviour in the event of a backplane bus error will have to be considered.

For the purpose of the following proofs and reasoning it will be assumed that systemError always reads NO_ERROR, and thus the following condition always holds

eq. 3 RECONFIGURE() = SUCCESS ⇔ check-reconfiguration(fID,...) = TRUE

meaning that function fID will reconfigure only if the result of the operation RECONFIGURE() (s.45) reads SUCCESS. From the definition of the operation RECONFIGURE() (s.45) this will be the case only if the function check-reconfiguration returns TRUE for the function fID (s.45.7) and systemError reads NO_ERROR (s.45.9).

### 8.2.1. Reconfiguration can only increase module function criticality

In this section it will be shown that the following property of the reconfiguration scheme is guaranteed:

P1. Function fID will reconfigure to function newID, only if the criticality of the latter function is higher than the criticality of fID, and the newID function is lost.

As discussed before, reconfiguration will be successful only if check-reconfiguration(fID,...) returns TRUE. Therefore, in order to guarantee that a more critical function will never reconfigure to a less critical one, it has to be shown that

eq. 4 RECONFIGURE() = SUCCESS ⇒ criticality(newID) > criticality(fID)

where fID and newID should be understood as in the specification (s.45). In order to avoid a situation where two core LRMs perform the same function after one of them had reconfigured to a function that has already been performed, it should be shown that the more critical function newID was lost prior to reconfiguration.

eq. 5 RECONFIGURE() = SUCCESS ⇒ in-lost(newID) = TRUE.

Assuming that no external errors occur (systemError = NO_ERROR), the LHS of eq. 3 can be substituted with its RHS into eq. 4, i.e.

eq. 6 check-reconfiguration(fID,...) = TRUE ⇒ criticality(newID) > criticality(fID)

The specification of check-reconfiguration(...) (s.40) states explicitly that function fID should only reconfigure if there exists a lost function f which is more critical than fID

eq. 7 $$\text{check-reconfiguration(fID,...)} = \text{TRUE} \Rightarrow$$

$$\exists\, f{:}ID \bullet \text{in-lost(f,...)} \wedge \text{criticality(fID)} < \text{criticality(f)}$$

As all non-negative entries in the strategy table are guarded by the criticality check in function get-level(...) (s.31.1), explicit criticality verification from eq. 7 (function check-reconfiguration(...) s.40.6) is also implicitly included in the statement from s.40.5 (see proof below). The inclusion of both the explicit and the implicit condition reduces the risk of invalid reconfiguration due to module data corruption (see section 8.7). Note also that whilst eq. 7 states that only a less critical function is allowed to reconfigure to a more critical one, eq.8 (s.40.5) verifies that function fID should reconfigure only if it was assigned as a backup for the lost function.[66]

In order to show equivalence between conditions from eq. 7 and s.40.6, it can be written from s.40.6

eq.8 $$\text{check-reconfiguration(fID,...)} = \text{TRUE} \Rightarrow$$

$$\exists\, f : ID \bullet \text{in-lost(f,...)} \wedge \text{strategy(entry(fID,f))} \neq -1$$

From the definition of the strategy look-up table (see functions do-strategy(), s.32, get-level(...), s.31, and also the invariant of the type Strategy, s.16), it follows that the relevant table entry is not equal to "-1" only if fID is less critical than f, i.e.

eq. 9 $$\text{strategy(entry(fID,f))} = \text{get-level(fID,f)}$$

eq. 10 $$\text{get-level(fID,f)} \neq -1 \Leftrightarrow \text{criticality(fID)} < \text{criticality(f)}$$

From equations eq. 9 and eq. 10 it can be seen that a non-negative entry in the strategy table[67] implies that function fID is of lower criticality than function f, i.e.

eq. 11 $$\text{strategy(entry(fID,f))} \neq -1 \Leftrightarrow \text{criticality(fID)} < \text{criticality(f)}$$

Substituting from eq. 11 into eq.8 proves the required equivalence of conditions stated in s.40.5 and s.40.6.

Taking into account eq. 7, eq.8 and eq. 11 it is now easy to show that the following equation holds

---

[66] As in static function based strategy tables each function provides backup for all more critical ones, both conditions will describe the same situation in this case.

[67] Note that all entries in the strategy table are greater or equal to "-1".

eq. 12 $\text{RECONFIGURE}() = \text{SUCCESS} \Leftrightarrow \text{check-reconfiguration}(\text{fID}) = \text{TRUE} \Rightarrow$

$\exists\, f{:}\text{ID} \bullet \text{in-lost}(f) \wedge \text{strategy}(\text{entry}(\text{fID},f)) \neq -1 \wedge \text{criticality}(\text{fID}) < \text{criticality}(f)$

This guarantees the following property P2:

P2. The reconfiguration conditions for the avionics function fID are only satisfied if there exists a function f, which is lost and more critical than fID.

To show that the property P1 holds, it is required to prove that the function to which fID reconfigures (newID) is more critical than fID (eq. 4).[68] From the definition of RECONFIGURE() (s.45.8), and from the specification of detect-failure(...) (s.41) it follows that

eq. 13 $\text{newID} = \text{detect-failure}(\text{fID},....)$ (s.45.8)

eq. 14 $\text{detect-failure}(\text{fID},...) = r \Rightarrow \text{strategy}(\text{entry}(\text{fID},r)) \neq -1$ (s.41.4)

eq. 15 $\text{detect-failure}(\text{fID},...) = r \Rightarrow \text{criticality}(\text{fID}) < \text{criticality}(r)$ (s.41.5)

eq. 16 $\text{detect-failure}(\text{fID},...) = r \Rightarrow \text{in-lost}(r,...)$ (s.41.2)

From equations eq. 11, eq. 13, eq. 14, eq. 15 and eq. 16 it can be now derived that the following statement evaluates to TRUE:

eq. 17 $\text{criticality}(\text{newID}) > \text{criticality}(f) \wedge \text{in-lost}(\text{newID},...)$

Thus, the new function (newID) is more critical than the one performed so far, and it has been lost prior to reconfiguration.

Equation eq. 17 in conjunction with the previously shown relations proves the required property P1, that function fID will reconfigure to function newID, only if the criticality of the latter function is higher than the criticality of fID, and the newID function is lost.

### 8.2.2. At least one of the function backups will reconfigure on the function loss

Assuming that a critical function is lost, and there exists a backup for this function, it has to be shown that at least one backup will reconfigure. This can be stated as the following property

[68] Note that equation eq. 12 states only that the reconfiguration conditions will be satisfied if there exists an appropriate lost function f, but it does not specify the function to which fID will reconfigure (newID).

P3. At least one of the function backups will reconfigure on the function loss.

Property P3 can be further expressed with the following equation (eq. 18)

eq. 18 $$\forall f{:}ID \bullet (\exists fID : ID \bullet strategy(entry(fID,f)) \neq -1) \bullet$$

$$\neg \text{ in-lost}(fID,...) \wedge \text{in-lost}(f,...) \Rightarrow$$

$$\exists \text{ time} : Time \bullet currentTime = time \wedge$$

$$check\text{-}reconfiguration(fID,....) = TRUE \wedge detect\text{-}failure(fID,...) = f$$

which reads that for each avionics function f which has got a backup (fID), the situation where f is lost and fID is operating, implies that there will be a point in time (currentTime = time), when the reconfiguration conditions will be satisfied for fID with respect to the lost function f. Note that should there be other backup modules, the time in which eq. 18 will hold for each of them will be different (based on the strategy table), and thus it will prevent simultaneous reconfiguration of multiple backup modules (see section 8.2.3 for further discussion).

According to the strategy look-up table a working module performing function fID is in backup for the function f if the relevant table entry is not equal to "–1", i.e.

eq. 19 $$strategy(entry(fID,f)) \neq -1 \wedge \neg \text{ in-lost}(fID,...)$$

The reconfiguration conditions will be satisfied (see definition of check-reconfiguration(), s.40) if both eq. 19 and eq. 20 hold, where eq. 20 reads

eq. 20 $$lost\text{-}delays(f,...) - 1 \geq strategy(entry(fID,f))$$

Note that as discussed before (eq. 11) the criticality check (s.40.6) will hold if eq. 19 holds. Note also that the fact that eq. 20 holds, will ensure that the routine in-lost(f,..) returns TRUE (see equations eq. 24 - eq. 26 for proof).

The above equation (eq. 20) states that the reconfiguration should occur if the function f is lost for the number of FAIL-DELAYs corresponding to fID backup level for f, and it can be rewritten from the definition of lost-delays(...) (s.38) as:

eq. 21 $$lost\text{-}delays(f,...) - 1 \geq strategy(entry(fID,f)) \Leftrightarrow$$

$$((time - responses(f)) \text{ div } delays(f)) - 1 \geq strategy(entry(fID,f))$$

From the specification of lost-delays(...) (s.38), it can be seen that as the system time elapses and the most recent responses from lost functions become "older" (no new messages are logged, so the time difference between the last stamped activity and the system clock increases), the return value of lost-delays(...) for those functions will also increase with every time interval of delays(f) (in this specification 50 ms for all functions - see FAIL-DELAY, s.2).

> Commentary:
> Since the reconfiguration scheme operates on static reconfiguration strategy data, the look-up table remains intact regardless of the system state. Thus the values of delays(f) and responses(f) do not change while the function in questions is lost and no new responses are recorded (see specification of the operation RECORD-RESPONSE(...), s.44). At the same time the time value (currentTime) continuously increases (e.g. every millisecond), so the LHS of equation eq. 21 also increases.

It has been assumed that the system time variable has no upper limit in terms of a single system run, and therefore it can be stated that at some point in time the following condition will hold

eq. 22 $\exists$ time:Time • ((time - responses(f)) div delays(f)) -1 $\geq$ strategy(entry(fID,f))

By a simple rearrangement of eq. 22 and from equation eq. 21, it can be now stated that eq. 20 will hold for a finite value of time:

eq. 23 $$\text{lost-delays}(f,...) - 1 \geq \text{strategy}(\text{entry}(fID,f)) \Leftrightarrow$$

$$\text{time} \geq (\text{strategy}(\text{entry}(fID,f)) + 1) \times \text{delays}(f) + \text{responses}(f)$$

From the definition of the strategy look-up table (see do-strategy(), s32), the function in-lost(...) (s.39), and based on the fact that eq. 20 holds, it is now relatively easy to show that the function f must be lost for the reconfiguration conditions to be satisfied. From the definition of the strategy table (function get-level(), s.31) it can be written that:

eq. 24 $$\text{strategy}(\text{entry}(fID,f)) \neq -1 \Leftrightarrow \text{strategy}(\text{entry}(fID,f)) \geq 0$$

From eq. 24 and eq. 22, and from the definition of the function in-lost(...) (s.39) it can be now stated that

eq. 25 $$\exists \text{ time} : \text{Time} \bullet$$

$$((\text{time} - \text{responses}(f)) \text{ div } \text{delays}(f)) - 1 \geq \text{strategy}(\text{entry}(fID,f)) \wedge \text{strategy}(\text{entry}(fID,f)) \geq 0 \Leftrightarrow$$

$$((\text{time} - \text{responses}(f)) \text{ div } \text{delays}(f)) \geq 1$$

eq. 26 $$((\text{time} - \text{responses}(f)) \text{ div } \text{delays}(f)) \geq 1 \Leftrightarrow \text{in-lost}(f,\ldots) = \text{TRUE}$$

At this point eq. 19, eq. 20 and eq. 26 hold, and thus the reconfiguration algorithm guarantees the property P4, stating that an existing operating backup will reconfigure in a finite time, providing that no system errors occur.

P4. If a backup for a lost function exists it will reconfigure in a finite time.

It needs now to be shown, that on reconfiguration the backup module will select the right function. Two cases have to be considered:

- function f is the only one that is lost and for which function fID is supposed to provide backup

eq. 27 $$\exists f : ID \wedge \neg \exists f_1 : ID \bullet f \neq f_1 \wedge \text{in-lost}(f_1,\ldots) \wedge \text{strategy}(\text{entry}(fID,f_1)) \neq -1 \wedge$$

$$\text{in-lost}(f,\ldots) \wedge \text{strategy}(\text{entry}(fID,f)) \neq -1$$

- reconfiguration conditions for function fID are satisfied with respect to multiple lost functions f, $f_1$, $f_2,\ldots,f_n$ (eq. 27 does not hold).

In the first case it is easy to prove, that the function detect-failure(fID,...) (s.41) will select the only lost function f. From the specification of detect-failure(fID,...) (s.41), it can be seen that conditions from lines s.41.2, s.41.3 and s.41.4 are satisfied respectively by equations eq. 26, eq. 20 and eq. 19. Note that the condition s.41.5 evaluates to s.41.4 and thus does not need to be considered separately (as discussed before it only reduces the risk of invalid reconfiguration due to data corruption). The condition expressed in lines s.41.6 - s.41.9 is satisfied for f, from the assumption (eq. 27) that there are no more lost functions for which fID has to provide backup.

In the second case[69] eq. 27 will not hold, and thus it is possible that one of the other functions ($f_1, f_2, \ldots, f_n$) will be selected for reconfiguration. This however, is guarded by the condition for the line s.41.9 to

[69] Such a situation could only occur during module/cabinet initialisation (see section 8.2.5), or if the system encountered multiple nearly simultaneous failures of processing modules (see also section 8.7.3).

enforce that one of the most critical functions will be selected, i.e. $f_1$, $f_2$, ..., $f_n$ must be of same or higher criticality than f. In the case where another function $f_x$ has been selected for reconfiguration, fID ceases to provide backup for f, and thus LHS of eq. 18 evaluates to false, which causes the implication (eq. 18) to evaluate to TRUE, and thus the property P3 is satisfied. Should f be one of the most critical functions it will either be selected and thus the scheme will satisfy the property P3, or alternatively the backup module will reconfigure to another equally critical function, what will cause the property P3 to be satisfied (see the discussion above).

Should the lost function f have more backups (i.e. fID was not the only operating backup), the backup modules will have to satisfy the property P3. As the number of functions in the cabinet is finite, three cases can be distinguished:

- for one of the backup functions eq. 27 will hold (i.e. f will be the only lost function for which it has to provide backup), and thus the property P3 will be satisfied as discussed beforehand,
- function f ill be of the highest criticality between the lost functions and it will be selected for reconfiguration (see section 8.2.4)
- all backups for f will reconfigure to other lost functions of same or higher criticality, therefore the reconfiguration scheme will operate adequately and the property P3 will be satisfied (as discussed above, f will have no operating backups and LHS of eq. 18 will yield FALSE and thus equation eq. 18 will evaluate to TRUE).

Thus the scheme guarantees the property P3, that at least one of the function backups will reconfigure on the function loss.

### 8.2.3. The least critical function will reconfigure first

In order to preserve the most critical functions, the reconfiguration scheme must ensure that in the event of a function loss the least critical backup module will reconfigure first. This can be stated as the following property:

P5. If there exist backup modules for a lost function, the module performing the least critical function will reconfigure first.

and can be formally described with the following equation

eq. 28 $$\exists f, f_1, f_2 : ID \bullet \text{in-lost}(f,...) \wedge \neg \text{in-lost}(f_1,...) \wedge \neg \text{in-lost}(f_2,...) \wedge$$
$$\text{strategy}(\text{entry}(f_1,f)) \neq -1 \wedge \text{strategy}(\text{entry}(f_2,f)) \neq -1 \wedge \text{criticality}(f_1) < \text{criticality}(f_2) \Rightarrow$$
$$\neg \exists t : Time \bullet \neg \text{check-reconfiguration}(f_1,...,t) \wedge \text{check-reconfiguration}(f2,...,t)$$

The left hand side of eq. 28 satisfies two (s.40.3, s.40.5) of the three conditions that must hold in order for the function check-reconfiguration to yield TRUE (note that both conditions are static, i.e. they do not depend on time). To prove property P5 it has to be shown, that the third condition (s.40.4, eq. 20) will be satisfied first for the less critical function. As the result of evaluation of the condition from s.40.4 depends clearly on the current time (see function lost-delays, s.38), and the system time is assumed to be an increasing natural number, in order to prove the property P5, it is sufficient to show that there does not exists a value of time for which the condition will be satisfied with respect to the more critical function, and yet it will not hold for the less critical one.

From the property P4, it is certain that there exist values of the time variable for which the reconfiguration conditions will be satisfied for functions $f_1$ and $f_2$. From eq. 23 the smallest value of time for which the condition will be satisfied for function $f_1$ can be written as follows,

eq. 29 $$t_1 = (\text{strategy}(\text{entry}(f_1,f)) + 1) \times \text{delays}(f) + \text{responses}(f)$$

and from the same equation, the value for function $f_2$ can be calculated as:

eq. 30 $$t_2 = (\text{strategy}(\text{entry}(f_2,f)) + 1) \times \text{delays}(f) + \text{responses}(f)$$

Again from equation eq. 23 it is known (provided the module has not yet reconfigured to perform a different function), all values of time greater than $t_1$ will satisfy reconfiguration conditions for function $f_1$, and analogously for $t_2$ and $f_2$.

To prove the required property P5 (eq. 28) it is now sufficient to show that $t_1$ is smaller than $t_2$. When backups for a given avionics function are being considered, it can be derived from the definition of the strategy look-up table (see do-strategy(), s.32, get-level(...), s.31, and has-continuous-columns(...), s.29), that the more critical the backup module function the higher its backup level. This can be formally expressed in the following equations:

eq. 31 $$\text{criticality}(f_1) < \text{criticality}(f_2) \Leftrightarrow id_1 > id_2$$

where $id_1$ and $id_2$ represent ID numbers of functions $f_1$ and $f_2$ respectively. From eq. 28, eq. 31 and the definition of function get-level(...), s.31, it can be now read that if both functions ($f_1$ and $f_2$) are in backup for function f (i.e. $\text{criticality}(f_1) < \text{criticality}(f)$ and $\text{criticality}(f_2) < \text{criticality}(f)$), the following equation will hold with respect to the relevant entries in the strategy table:

eq. 32 $$\text{strategy}(\text{entry}(f_1,f)) = \text{maxID} - id_i \wedge \text{strategy}(\text{entry}(f_2,f)) = \text{maxID} - id_2 \Rightarrow$$
$$\text{strategy}(\text{entry}(f_1,f)) < \text{strategy}(\text{entry}(f_2,f))$$

Therefore, from eq. 32 and the properties of the natural numbers (all look-up table entries are natural numbers) it can be derived that

eq. 33 $$\exists\, i \in N \bullet i > 0 \wedge \text{strategy}(\text{entry}(f_2,f)) = \text{strategy}(\text{entry}(f_1,f)) + i$$

Substituting the term $\text{strategy}(\text{entry}(f_2,f))$ from eq. 33 to eq. 30, $t_2$ can be expressed as

eq. 34 $$t_2 = (\text{strategy}(\text{entry}(f_1,f)) + i + 1) \times \text{delays}(f) + \text{responses}(f) \wedge i > 0$$

and further evaluation of the time difference $t_2 - t_1$ based on equations eq. 29 and eq. 34, gives the following result

eq. 35 $$t_2 - t_1 = (\text{strategy}(\text{entry}(f_1,f)) + i + 1) \times \text{delays}(f) + \text{responses}(f) -$$
$$(\text{strategy}(\text{entry}(f_2,f)) + 1) \times \text{delays}(f) + \text{responses}(f) \wedge i > 0 \Leftrightarrow$$
$$t_2 - t_1 = i \times \text{delays}(f) \wedge i > 0$$

Following the fact that delays are implemented as positive natural numbers (see s.2, s.7, s.19.4, s.19.11 and s.35 in the specification), the following equation holds for all entries in the delay array (s.19.4).

eq. 36 $$\forall\, f{:}ID \bullet \text{delays}(f) > 0$$

When combined with the above statement (eq. 36), equation eq. 35 shows that $t_2$ is greater than $t_1$, and therefore check-reconfiguration(...) will yield TRUE first for $f_1$, and only later for $f_2$. This in view of equation eq. 23, proves the required property P5 (eq. 28).

Commentary:
Clearly if check-reconfiguration($f_1$,...,$t_1$) holds for all values of time greater or equal to $t_1$ and $t_2$ is greater than $t_1$, it is impossible for check-reconfiguration($f_2$,...,t) to hold and yet check-reconfiguration($f_1$,...,t). Note that should t be greater than $t_2$ it will also be greater than $t_1$.

It has to be noted that the time difference $t_2$ - $t_1$ needs to be long enough to allow the less critical module to restore execution of function f, so that function $f_2$ will not reconfigure (i.e. both functions must not reconfigure to f). If $f_1$ reconfigures successfully to f, in-lost(f,...) will yield FALSE, and thus check-reconfiguration($f_2$,...) will also yield FALSE with respect to function f. In order to ensure that such time requirements are met, the appropriate value of delays(f) must be chosen for each avionics function which is clearly dependent on the system being integrated.

Should the delays be sufficiently long and provided that the failure detection mechanism works properly (see section 8.5 for discussion on the failure detection mechanism), it is easy to show that the following property of the scheme is guaranteed:

P6. Only one function will reconfigure in the event of any single function loss.

This can be proven in a relatively straightforward manner. Properties P4 and P5 guarantee appropriate selection of a backup module to reconfigure on a function loss. To show that only this one (least critical backup) core LRM will reconfigure it has to be shown that none of the modules not in backup will perform reconfiguration. This again follows the definition of function check-reconfiguration(...) (s.40) and its condition expressed in s.40.5. The condition states the function will be allowed to reconfigure only if it is in backup for the lost application (relevant entry in strategy table not equal to "-1"), which guarantees property P6a:

P6a. None of the modules not in backup for the lost function will reconfigure.

Properties P4, P5 and P6a guarantee that only one function will reconfigure in the event of any single function loss[70].

[70] Assuming again reliable fault detection and no system errors.

### 8.2.4. On recovery a module will reconfigure to the most critical function not being performed

On system start-up and on module recovery from a transient fault each core LRM selects one of the lost functions to perform. The selection mechanism is based on monitoring of the backplane bus to identify functions currently performed in the cabinet, and it treats recovery and cabinet initialisation in a uniform manner. However, it is important that the following property is guaranteed by the scheme, when the function selection mechanism is being considered:

P7. On initialisation (system start-up or recovery) a module will select the most critical function not performed in the cabinet, and it will resume its execution.

As read in the operation MAIN()[71] (s.50) each module initially monitors the backplane bus for a period of time dependent on the value of RECOVERY-DELAY and the module ID. After that time core LRMs select from the lost functions the one with lowest the ID (function get-id(...), s.49). Such behaviour not only ensures that one of the most critical function will be selected, but it also guarantees the order of selection of functions of same criticality (deterministic operation).

Commentary:
The operation of such a mechanism is based on increasing data bus monitoring intervals related to increasing core LRM IDs. Thus, the most critical function module will initially monitor the backplane bus for the shortest time interval, that will allow it to perform function selection first, and consequently to select the most critical function with the lowest function ID. When the next module finishes its monitoring of the backplane bus, it will have observed that the most critical function is already being performed, and it will select the second most important function. It becomes clear that the RECOVERY-DELAY must be long enough to allow a core LRM to resume execution of an avionics function, so that all the remaining modules are able to detect the data bus activity from said function.

In order to be able to properly identify lost functions, the module must have monitored the backplane bus for a sufficiently long time. It can be read from the specification of function in-lost(....), s.39, that function f is announced lost if the following equation holds:

eq. 37 $$\text{in-lost}(f,...) = \text{TRUE} \Leftrightarrow \text{lost-delays}(f,....) \geq 1$$

This can be rewritten from the specification of function lost-delays, s.38, as

[71] MAIN() is assumed to be the first operation the module will execute after initialisation of the state - init rtma

eq. 38 $\quad$ in-lost(f,...) = TRUE $\Leftrightarrow$ ((time - responses(f)) div delays(f)) $\geq$ 1

The condition of eq. 38 will be satisfied for every avionics function in the cabinet (should it be lost), if the value of time is sufficiently greater than the value of responses(f). The smallest sufficient time value ($time_f$) can be easily calculated from eq. 38

eq. 39 $\quad$ $\forall$ f:ID • $time_f \geq$ delays(f) + responses(f)

Taking into account that on module initialisation all values in the responses array will be equal to zero (initial time), eq. 39 can be further simplified as:

eq. 40 $\quad$ $\forall$ f:ID • $time_f \geq$ delays(f)

The point in time when the module performs its function selection ($time_s$), depends on the value of the RECOVERY-DELAY and the module ID (s.50.4)

eq. 41 $\quad$ $time_s \geq$ startTime + (currentID + 1) $\times$ RECOVERY-DELAY

Taking into account that on module initialisation startTime is reset to zero and combining equations eq. 41 and eq. 40, it must be shown that for each module and each function $time_s$ is greater or equal to $time_f$, that can be formally written as follows

eq. 42 $\quad$ $\forall$ fID, f : ID • (fID + 1) $\times$ RECOVERY-DELAY $\geq$ delays(f)

Equation eq. 42 will hold for every fID, if it holds for the lowest one, i.e. if eq. 42 holds for the lowest value of LHS it will always hold. Therefore eq. 42 can be used to calculate a suitable value of RECOVERY-DELAY

eq. 43 $\quad$ $\forall$ f : ID • RECOVERY-DELAY $\geq$ delays(f).

Thus the value of RECOVERY-DELAY must be at least equal to the longest FAIL-DELAY. Provided this is the case (the specification satisfies this condition), it is guaranteed that all modules will be able to detect the lost functions during initialisation, and the definition of function get-ID() (s.49) guarantees that the will select the most critical one.

### 8.2.5. After initialisation / recovery all modules will perform functions with unique IDs

As discussed above, the mechanism for function selection guarantees that a module will resume execution of the lost functions with the lowest ID. From the specification of operation MAIN(), s.50, it can be seen that each module determines the time for selection of an avionics function based on its ID and the RECOVERY-DELAY, s.50.4. Specifically, there is the RECOVERY-DELAY period of time between two modules with consecutive IDs attempt to select their functions. Provided that the RECOVERY-DELAY is long enough to allow restoration of function execution (this can be safely assumed, although the exact length is clearly system and implementation dependent), the following property is guaranteed:

P8. On initialisation all modules will select unique functions to execute.

> Commentary:
> If module fID selects the most critical lost function f and starts to perform it, before the next RECOVERY-DELAY elapses data related to this function will be present on the backplane bus. Therefore, the next module fID+1 will not perceive f as lost, and it will select the next available function.

Note, that in the case of multiple nearly simultaneous recoveries this property cannot be guaranteed, as modules of different IDs could start monitoring of the data bus at such points in time, that their selection routines will overlap in an undesirable manner. The probability and consequences of nearly simultaneous events are further discussed in section 8.7.3.

### 8.2.6. Reconfiguration scheme will not cause reconfiguration (loss) of a critical function

As the objective of the scheme is to preserve critical functions, the following property should be required from the reconfiguration scheme.

P9. Reconfiguration scheme will not cause reconfiguration (loss) of a critical function.

The previously proven property P1 guarantees that a module will reconfigure only to a more critical function. Also, from previously proven equation eq. 12 it is certain that function $f_i$ will reconfigure only if

eq. 44 $$\exists f_2 \bullet \text{in-lost}(f2) \wedge \text{criticality}(f_2) > \text{criticality}(f_1)$$

In the case where $f_1$ is one of the critical functions (highest criticality) eq. 44 will never hold simply because there does not exist a more critical function in the system, and thus property P9 is guaranteed.

## 8.3. Determinism of the Scheme

The notion of determinism in RIMA has been discussed in detail in Chapter 5. At this place the property of normal determinism will be required from the scheme[72], and it will be shown that the specified reconfiguration scheme meets the relevant requirements.

### 8.3.1. Definition

The normal determinism of a reconfiguration method ensures that the assignment of avionics functions to core LRMs depends solely on the sequence of encountered non-simultaneous system events (either failure or recovery), and not on their timing. Two events are thought to be non-simultaneous if the time interval between them is long enough to allow the system to reach a stable state after encountering the first event, and before the second one occurs. The system state is considered stable when none of the working core LRMs is expected to reconfigure or is being reconfigured, i.e. the previous event has been dealt with thoroughly and no further actions related to the event are required.

Before the proof can be conducted the state of the cabinet has to be defined. In order to preserve consistency with the discussion so far, a VDM based notation will be used. The state of the cabinet can be then defined in terms of a set of processing modules, each of which executes the specified reconfiguration scheme, and it differs from all the other core LRMs by the assigned avionics function (i.e. currentID) and its hardware ID (startID). Such cabinet state denoted by S can be formalised as:

eq. 45 $$S = \{m_0,\ldots,m_{maxID}\}$$

where $m_n$ refers to n-th module. In order to be able to operate on such defined state, the following notation will be used where $m_n$.*function-A(parameter-list)* denotes the call to *function-A* in the state of module $m_n$, and $m_n$.*state-variable* denotes the value of the *state-variable* in the state of core LRM $m_n$.

---

[72] The requirement for extended determinism would lead to a greatly increased complexity of the algorithm

A simple approach to the definition of a stable cabinet state could be based on the result of function check-reconfiguration for each core LRM. Should the function return FALSE for each working module and each avionics function, then none of the modules is expected to reconfigure. This could be formalised as follows:

eq. 46 $m_k$ is working $\Leftrightarrow$ $m_k$.systemError = <NO_ERROR>

i.e. a module is considered working if its systemError state variable reads NO_ERROR, and

eq. 47 S is stable $\Leftrightarrow \forall\ m \in S \bullet$ m is working $\Rightarrow$ m.check-reconfiguration(m.currentID,...) = FALSE.

Although this seems to be intuitively correct, eq. 47 yields a problem when recovery of processing modules is allowed. During recovery a core LRM monitors the backplane bus to identify the function it is supposed to reconfigure to. At the same time, the table containing time stamps of responses from avionics functions will be reset to the initial time (init rima, s.19.11, do-responses(...), s.34), and therefore check-reconfiguration will return FALSE even though the module may have to reconfigure shortly. To eliminate this problem, the stable state can be defined as a state where each working module does not see the need for reconfiguration (check-reconfiguration = FALSE), and none of the core LRMs is in the initialisation phase.

eq. 48 S is stable $\Leftrightarrow \forall\ m_1 \in S \bullet$

$m_1$ is working $\Rightarrow$ $m_1$.check-reconfiguration($m_1$.currentID,...) = FALSE $\wedge$

$\neg\ \exists\ m_2 \in S \bullet (m_2.\text{currentTime} \leq m_2.\text{startTime} + (m_2.\text{currentID}+1) \times \text{RECOVERY-DELAY})$.

However, this definition does not include the timing requirements related to multiple backups, as even though a function is lost and it has got a backup module, check-reconfiguration can return FALSE for this module, provided that the function has not been lost not lost for long enough (a lower backup level may already be unavailable and a higher backup level will not yet have to reconfigure, s.41.3). Therefore the definition of the stable state needs to be changed to

eq. 49

$$S \text{ is stable} \Leftrightarrow (\forall\, m_1 \in S \bullet m_1 \text{ is working} \bullet$$

$$\neg \exists\, f \in ID \bullet m_1.\text{in-lost}(f,...)=\text{TRUE} \wedge m_1.\text{strategy}(\text{entry}(m_1.\text{currentID},f)) \neq -1) \wedge$$

$$(\neg \exists\, m_2 \in S \bullet m_2 \text{ is working} \wedge$$

$$(m_2.\text{currentTime} \leq m_2.\text{startTime} + (m_2.\text{currentID}+1) \times \text{RECOVERY-DELAY})$$

With the assumption that no two events are simultaneous or nearly simultaneous, to show that the scheme exhibits the property of normal determinism, it now suffices to prove that an event occurring in the stable state $S_1$ will always transform this state into a stable state $S_2$ regardless of its time of occurrence. Let $S(e_t)$ denote a state obtained from inserting an event *e* into a stable state *S* at the time *t*. The scheme determinism can be written as follows:

eq. 50

$$\forall\, t : \text{Time} \bullet S_1(e_t) = S_2 \wedge S_1 \text{ is stable} \wedge S_2 \text{ is stable}$$

## 8.3.2. Proof of scheme determinism

In order to show scheme determinism one has to consider three cases:

- an event of module recovery occurs in a stable state
- an event of module failure occurs in a stable state
- a cabinet is initialised on system power-up.

To prove scheme normal determinism, it suffices to show that the initial state of the cabinet is stable, and that regardless of the timing of subsequent non-simultaneous events the same state transitions will be taken.

### 8.3.2.1. Determinism on module recovery

It is assumed that before a processing module recovers, the cabinet is in a stable state. Since recovery does not lead to a loss of any avionics function, all working core LRMs excluding the recovering one will have already met the state stability conditions, and will not attempt reconfiguration.

> Commentary:
> The return value of the in-lost routine depends solely on the set of lost functions. If an avionics function is lost, sufficiently critical and the module is expected to provide backup for this function, the first condition for the stability of the cabinet state will not be satisfied (see eq. 49). As the cabinet is assumed to be in a stable

state prior to module recovery, clearly neither of the lost functions is sufficiently critical. As the removal of a function from the set of lost applications (the recovering modules will select the most critical one) does not affect the criticality of remaining lost functions, all the modules working before the event will meet the stability condition after the recovery of a core LRM is completed.

As all the modules working prior to module recovery already satisfy the stability conditions, to show that the resultant state is stable, it suffices to show that regardless of the time of recovery, the recovering module will always select the same most critical function to reconfigure to.

From the property P7 it is known, that the recovering module will select the most critical function not being performed in the cabinet. From the discussion given in section 8.2.4 and 8.2.5 it is also known, that the recovering/initialising module will always select the function with the lowest ID that is not being performed in the cabinet. As recovery always occurs in the same stable state $S_1$[73], the new function will always be selected from the same set of applications and therefore the recovery/initialisation algorithm guarantees that the same function will be chosen regardless of time.

Commentary:
This is guaranteed by the time independent specification of the function get-id() (s.49), which selects the lowest ID function from the lost ones. The reliability of the failure detection mechanism is of a great importance (the module must be able identify the lost functions properly), but at this point the detection mechanism is assumed to be 100% reliable (discussion on operation with failure detection problems is given in section 8.5).

From the definition of MAIN (s.50), the second part of the stability condition for the state after recovery is straight forward true (it states that no module is in the initialisation stage, and as the recovering module selects and resumes its function, and no other events are allowed to occur, this will be true for all modules). It needs showing that the first part of the condition will hold as well.

The set of working modules ($C_r$) present in the state after recovery ($S_r$) can be written as the set of working modules (C) in the state beforehand (S), with additional newly recovered module $m_r$, i.e.

[73] This can be any stable state. The relation between the stable state before the event and the stable state after the event needs to be preserved for the scheme to be deterministic (see equation eq. 50).

eq. 51 $$C_r = C \cup \{m_r\}.$$

It has to be shown that after the recovery is completed the second part of stability condition holds, i.e. none of the modules should recognise the need for reconfiguration

eq. 52 $$\forall\, m \in C_r \bullet \neg\, \exists\, f \in ID \bullet m.\text{in-lost}(f,...)=\text{TRUE} \wedge$$

$$m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

Note the lack of the condition checking whether the module m is working, based on the definition of $C_r$ as the set of all working modules.

From the function selection algorithm the following relations between sets of lost functions prior recovery (L), and post recovery ($L_r$) can be easily observed:

eq. 53 $$L = L_r \cup m_r.\text{currentID}$$

eq. 54 $$\forall\, f \in L \bullet \text{criticality}(f) \leq \text{criticality}(m_r.\text{currentID})$$

eq. 55 $$\forall\, f \in L_r \bullet \text{criticality}(f) \leq \text{criticality}(m_r.\text{currentID})$$

the selected function is one of the most critical functions that were lost prior to the event, and yet it is one of the least critical that are performed post-recovery. Note that from the definition of function in-lost() (s.39), the following equations also hold:

eq. 56 $$\forall\, m \in C_r \bullet f \in L_r \Leftrightarrow m.\text{in-lost}(f...) = \text{TRUE}$$

eq. 57 $$\forall\, m \in C \bullet f \in L \Leftrightarrow m.\text{in-lost}(f...) = \text{TRUE}$$

Thus using the above relations and the definition of the function in-lost() (s.39), equation eq. 52 can be rewritten as follows:

eq. 58 $$\forall\, m \in C \cup m_r \bullet \neg\, \exists\, f \in ID \bullet f \in L_r \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

From the stability of state S, and from the assumptions of reliable fault detection and non-simultaneous events (i.e. other core LRMs must not reconfigure in the meantime), it is guaranteed that this condition will hold for all modules working prior to recovery. This can be shown as follows:

eq. 59 $$S \text{ is stable} \Leftrightarrow \forall\, m \in S \bullet m \text{ is working} \bullet$$

$$\neg \exists\, f \in ID \bullet m.\text{in-lost}(f,\ldots)=\text{TRUE} \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

eq. 60 $$S \text{ is stable} \Leftrightarrow \forall\, m \in S \bullet m \text{ is working} \bullet$$

$$\neg \exists\, f \in ID \bullet f \in L \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

eq. 61 $$S \text{ is stable} \Leftrightarrow \forall\, m \in C \bullet$$

$$\neg \exists\, f \in ID \bullet f \in (L_r \cup \{m_r.\text{currentID}\}) \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

eq. 62 $$S \text{ is stable} \Leftrightarrow \forall\, m \in C \bullet \neg \exists\, f \in ID \bullet$$

$$((f \in L_r \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1) \vee$$

$$(f = m_r.\text{currentID} \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1))$$

If function f invalidation the equation does not exist in the set $L_r \cup \{m_r.\text{currentID}\}$, it clearly does not exist in its subset $L_r$, i.e.

eq. 63 $$S \text{ is stable} \Rightarrow \forall\, m \in C \bullet \neg \exists\, f \in ID \bullet f \in L_r \wedge$$

$$m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

The above equation (eq. 63) shows that equation eq. 52 holds for all modules working prior to recovery.

To prove that the state $S_r$ is stable, it is now necessary to show that the condition expressed in equation eq. 52 holds for $m_r$ ($m=m_r$), i.e.

eq. 64 $$\neg \exists\, f \in ID \bullet f \in L_r \wedge m_r.\text{strategy}(\text{entry}(m_r.\text{currentID},f)) \neq -1$$

From the definition of the strategy table and equation eq. 11 the following equation holds

eq. 65 $$m_r.\text{strategy}(\text{entry}(m_r.\text{currentID},f)) \neq -1 \Leftrightarrow$$

$$\text{criticality}(m_r.\text{currentID}) < \text{criticality}(f)$$

Assuming that there exists function F which invalidates equation eq. 64, i.e. $S_r$ is not stable. Then

eq. 66 $$F \in L_r \wedge m_r.\text{strategy}(\text{entry}(m_r.\text{currentID},F)) \neq -1$$

Clearly equations eq. 55, eq. 65 and eq. 66 contradict, and therefore the assumption that function F exists cannot be true. Combining equations eq. 56, eq. 63 and eq. 64 it is easy to observe that the initial equation eq. 52 holds, and hence the state after recovery is stable. This, in conjunction with the previous discussion on deterministic function selection on recovery (see function get-id(), s.49), shows that equation eq. 50 will be satisfied if an event of module recovery occurs at any point in time in a stable state. This proves scheme normal determinism in the event of core LRM recovery.

#### 8.3.2.2 Determinism on module failure

To prove scheme determinism on module failure it has to be shown, that if reconfiguration is required to sustain a function lost in a given stable state, it will always affect the same core LRM (regardless of the time of failure occurrence), and that the resultant state will be stable. The proof will be conducted for two cases, first where the failed module was performing an avionics function without backup (i.e. reconfiguration was not required), and second where reconfiguration was required.

***Failure of a module without backup***

In the case of a failure of least critical module, none of the remaining working modules will reconfigure (this is guaranteed by property P2), and therefore the cabinet will behave deterministically regardless of timing (no action is always deterministic). In order to prove scheme determinism on such a module failure, it now suffices to show that in this situation the new state is stable. Clearly the second part of the stability condition holds, as none of the modules is recovering. Thus it is only necessary to show that none of the modules is required to reconfigure, i.e.

eq. 67 $$\forall\, m \in S_f \bullet m \text{ is working} \bullet$$

$$\neg\, \exists\, f \in \text{ID} \bullet m.\text{in-lost}(f,\ldots)=\text{TRUE} \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

where $S_f$ denotes the state after the failure. This is guaranteed by the property P1, and can be formally justified as follows.

Let S denote the state before failure, C the set of modules working in the state S, L the set of lost functions before the failure, $L_f$ the set of the lost functions after the failure of the module $m_f$, and $C_f$ the set of modules working after the failure. The following relations are clearly true:

eq. 68 $$C_f = C \setminus \{m_f\}$$

eq. 69 $$L_f = L \cup \{m_f.\text{currentID}\}$$

It is assumed that the state prior to the failure was stable, i.e.

eq. 70 $$\forall\, m \in S \bullet m \text{ is working} \bullet$$

$$\neg \exists\, f \in ID \bullet m.\text{in-lost}(f,...)=\text{TRUE} \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

This can be rewritten from the definition of function in-lost(...) (s.39) and using equations eq. 68 and eq. 69 as:

eq. 71 $$\forall\, m \in C_f \cup \{m_f\} \bullet$$

$$\neg \exists\, f \in ID \bullet f \in L_f \setminus \{m_f.\text{currentID}\} \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

If the above condition holds for every core LRM including $m_f$, it will hold for all of them excluding $m_f$. Thus it can be now written, that:

eq. 72 $$\forall\, m \in C_f \bullet$$

$$\neg \exists\, f \in ID \bullet f \in L_f \setminus \{m_f.\text{currentID}\} \wedge m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

The above equation (eq. 72) guarantees stability of $S_f$ for all the modules working in this state, and for all functions but the newly lost one ($f \in L_f \setminus \{m_f.\text{currentID}\}$). To show the complete stability of the state $S_f$ it now suffices to show that the state is stable with respect to the lost function (i.e. none of the modules sees the need for reconfiguration):

eq. 73 $$\forall\, m \in S_f \bullet m \text{ is working} \bullet m.\text{strategy}(\text{entry}(m.\text{currentID},m_f.\text{currentID})) = -1$$

As the lost function ($m_f$.currentID) was assumed to be of the lowest criticality in the system, it is clearly less critical than functions performed by all core LRMs, i.e.

eq. 74 $$\forall\, m \in C_f \bullet \text{criticality}(m.\text{currentID}) \geq \text{criticality}(m_f.\text{currentID})$$

Again from equation eq. 11 it is known, that

eq. 75 $$\text{strategy}(\text{entry}(\text{currentID},f)) \neq -1 \Leftrightarrow \text{criticality}(\text{currentID}) < \text{criticality}(f)$$

It is now clear that, since each module function is of equal or higher criticality than the lost function, equation eq. 73 holds, which combined with equation eq. 72 guarantees that the state $S_f$ is stable.

***Failure of a module with backup***

So far the scheme guarantees deterministic and stable behaviour when the system encounters a failure of a core LRM performing one of the least critical functions. It is now required to show, that the scheme will behave deterministically also when reconfiguration is necessary in order to sustain the temporarily lost application. This can be achieved by showing that, regardless of the time of failure, the same module will reconfigure to the lost function, and that the resulting state is stable.

The already proven property P5 guarantees that the least critical backup will reconfigure first on the function loss. Moreover, property P6 ensures that only the least critical backup will reconfigure, if a backup exists. These two properties (together with the property P4, which guarantees that if there exists a backup it will reconfigure in a finite time) are sufficiently strong to ensure that the same module will reconfigure on a failure of an avionics function with a backup, should it be encountered in the same stable state.

The proof that the same lost function will be chosen by the backup module to reconfigure to is also relatively straightforward. Assuming that the state before the failure (S) was stable, none of the working modules (let C denote the set of working modules prior to failure) observed the need for reconfiguration, i.e. none of the functions being lost (let L denote set of lost functions prior to the event) was of a sufficiently high criticality, after the failure (but prior to reconfiguration) the set of lost functions ($L_f$) can be described as

eq. 76 $$L_f = L \cup \{m_f.\text{currentID}\}$$

where $m_f$ denotes the failed module. As the state before the failure is assumed to be stable, none of the functions in L has got a backup module that is supposed to reconfigure, therefore the only function for which the reconfiguration conditions can be satisfied is the lost one ($m_f$.currentID). As has been already assumed, in this case the lost function has got a backup (the case where a function with no backup had been lost has been discussed in section 8.3.2.2.1), the property P4 guarantees that the backup will reconfigure. Thus the properties P4, P5 and P6, in conjunction with the stability of the state prior to the failure, guarantee that the same module will reconfigure and to the same function, regardless of the time of failure.

To complete the proof of scheme determinism on a module failure, it has to be shown that the resultant state is stable. Let $m_r$ denote the reconfiguring module, and $S_r$ denote the state after reconfiguration (the state after the failure $S_f$ is of little interest, as it is by definition transitory and unstable). The following equations describe relations between the set of working modules (C), the set of lost functions (L) prior to the failure, the set of working modules ($C_r$) and the set of lost functions ($L_r$) post-reconfiguration.

eq. 77 $$C_r = C \setminus \{m_f\}$$

eq. 78 $$L_r = L \cup \{m_r.\text{currentID}\}$$

Note that $m_f$.currentID is not lost, as the processing module $m_r$ reconfigured to sustain its execution.

In order to show the stability of the state $S_r$, it is required to show that the equation below holds.

eq. 79 $$\forall\, m \in S_r \bullet m \text{ is working} \bullet$$

$$\neg \exists\, f \in ID \bullet m.\text{in-lost}(f,...)=\text{TRUE} \wedge m.\text{strategy}(\text{entry}(m_1.\text{currentID},f)) \neq -1$$

Note that the second part of the stability condition from equation eq. 49 is not included, as it is clearly true since none of the modules are recovering (simultaneous events are not permitted), and the cabinet is not in the power up sequence (this case is discussed separately in the following section).

From the stability of the state S prior to the event, it is clear that the following equation holds

eq. 80 $$\forall\, m \in C \bullet \neg \exists\, f \in L \bullet m.\text{strategy}(\text{entry}(m.\text{currentID})) \neq -1$$

which can be rewritten with the use of equations eq. 77 and eq. 78 as

eq. 81 $$\forall\, m \in C_r \cup \{m_f\} \bullet$$

$$\neg \exists\, f \in L_r \setminus \{m_r.\text{currentID}\} \bullet m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

Again, if the above equation holds for all modules from the set $C_r \cup \{m_f\}$, it will hold for the less general case based on the set $C_r$, i.e. the following equation is clearly true

eq. 82 $$\forall\, m \in C_r \bullet \neg \exists\, f \in L_r \setminus m_r.\text{currentID} \bullet m.\text{strategy}(\text{entry}(m.\text{currentID},f)) \neq -1$$

To complete the proof it is necessary to show that equation eq. 79 holds with respect to function $m_r$.currentID, i.e. it has to be shown that the following equation holds:

eq. 83 $\forall\ m \in C_r \bullet m.strategy(entry(m.currentID, m_r.currentID)) \neq -1$

From equation eq. 11 it is known that the relevant entry(currentID,f) in module strategy table will not be equal to "-1", only if the function f is more critical than the module function described by currentID. Therefore, it has to be shown that the reconfigured module had been performing one of the least critical functions prior to the event.

From the definition of the strategy table it is known that a module provides backup for all more critical functions and not for any less critical ones. This combined with the properties P5 and P6 guarantees that $m_r$.currentID was one of the least critical functions being performed in the cabinet before the failure, and thus it shows that equation eq. 83 holds.

The above equations eq. 82 and eq. 83 show that equation eq. 79 holds, and thus the resultant state is stable. The scheme normal determinism in the event of a failure of a module with backup is guaranteed.

#### 8.3.2.3. Determinism on initialisation

The deterministic and stable behaviour of the reconfiguration scheme - as specified in section 8.3 - has been proven with respect to non-simultaneous events of failure and recovery occurring in any stable state. In order to assure scheme determinism in any sequence of events, it has to be shown that the system will reach a stable state on initialisation. As one of the objectives of RIMA is achieving high tolerance for dispatch with a number of failed modules[74], the considerations in this section must not focus only on the case where all modules are operational.

As discussed in section 8.2.5, the initialisation mechanism ensures that the module selects the lowest ID function from the set of lost applications (i.e. one of the most critical ones). The algorithm guarantees also that all modules will select unique functions (see property P8), and therefore it will assign N most critical functions to N working modules. It is now relatively easy to notice, that after initialisation the resultant state is stable.

---

[74] This may rise various safety related issues, as current CAA/FAA regulations forbid aircraft dispatch with a known failure.

Commentary:
Clearly the second part of the stability condition holds, as all working modules have completed their initialisation.

Let C denote the set of modules expected to work after initialisation, and L denote the set of lost functions. The required stability equation takes the following form

eq. 84 $\forall$ m in C • $\neg \exists$ f $\in$ L • strategy(entry(m.currentID,f)) $\neq$ -1

Again from equation eq. 11 it is known, that the module will constitute a backup for function F if and only if F is of higher criticality than its current function

eq. 85 strategy(entry(currentID,f)) $\neq$ -1 $\Leftrightarrow$ criticality(currentID) < criticality(f)

However, since N most critical functions are already assigned to N working modules, all functions in set L are of at most equal criticality to the performed applications, and none is of higher. Therefore, the state after the initialisation is completed has to be regarded as stable.

Having shown that the system initialises to a stable state, and consecutive non-simultaneous events of failure and recovery deterministically transform one stable state into another regardless of the event timing, the required determinism of the reconfiguration scheme is guaranteed. This has been achieved on the assumption that events do not occur simultaneously or nearly simultaneously, and that the failure detection mechanism operates without faults.

Although the probability of nearly simultaneous events is extremely low (they would have to happen within the time span of at most several hundreds of milliseconds), the consequences of such improbable coincidences are discussed in section 8.7. Also the scheme operation with the faulty event detection mechanism is discussed in section 8.5 of this chapter.

At this point normal determinism of the reconfiguration scheme is guaranteed under reasonable assumptions of reliable failure detection and non-simultaneous events.

## 8.4. Validity of the reconfiguration strategy data throughout the system lifespan

As previously discussed, the presented reconfiguration scheme is based on static strategy data, that does not undergo any changes as the system evolves. However, in order to show that the data remains valid at any point of time, it is necessary to show that the system will not evolve in a manner, that could invalidate the strategy table.

In this approach the reconfiguration data will be considered invalid if, despite reliable fault detection and no external data corruption[75], it allows one of the following undesirable types of behaviour:

- it allows two modules to reconfigure to the same function
- prevents a non-critical module from reconfiguration to a more critical lost function
- a backup module reconfigures, but the time taken for reconfiguration exceeds the system permitted limits.

The third situation is relatively easy to discuss. Provided that the fault detection mechanism operates correctly, the time a core LRM waits before reconfiguring depends solely on its backup level (relevant strategy table entry). The strategy table ensures that the time constraints are met for the highest backup level in the cabinet[76], i.e. for the module which will be required to reconfigure last. Therefore a module will exceed the system time limits on reconfiguration if at least one entry in the reconfiguration data changes from the original or the failure detection mechanism fails. This is not considered as data invalidity, and operation with corrupted data or with unreliable fault detection mechanism is discussed later in this chapter.

The second case in which the data might become invalid relates to a situation where a non-critical core LRM fails to provide backup for a more critical function. With respect to the strategy table, this can only be the case if a relevant data entry reads "-1", instead of describing a non-negative backup level. As the

---

[75] For example, a hardware memory fault could lead to data changes, that would not be considered as strategy data invalidity, as such events are thoroughly independent from the scheme. Other means, such as multiple copies of the data or highly reliable hardware must be used to deal with such situations.

[76] The time constraints conformance is discussed in detail in section 8.6.

initial data is free of such problems, a core LRM will fail to provide backup if at least one of its backup levels has been changed to "-1". It has been discussed before that the presented reconfiguration scheme is based on static data (no changes to the reconfiguration table are performed), and therefore a processing module may fail to provide backup only if an event external to the scheme has altered the look-up table. This, however, is not considered as data invalidity in this section.

The first of the problems listed above refers to the situation where two modules are in the same backup level for the same function. As all non-negative entries in each column are unique (provided no changes occurred to the data), two core LRMs can be in the same backup level for a function, if their currently performed functions are identical. Clearly, this is not the situation after system initialisation (see property P8), and therefore it would have to happen during the cabinet operation.

It is easy to notice, that two modules can perform identical functions if:

- a problem with the fault detection mechanism indicated to a core LRM a need for reconfiguration to an avionics function that was not lost, and thus the module has reconfigured to a function already being performed by another core LRM
- two modules have already reconfigured to the same lost function.

The first case will not be considered in this section as it requires unreliable failure detection mechanism (consequences of such behaviour are discussed in section 8.5). The second case requires two modules to have reconfigured before to the same function *F1*, in order to perform later the same function *F2*. However, applying recursively the same reasoning to this situation leads to the observation that two modules will reconfigure to the same function only if they have already reconfigured to a different function (the same application for both core LRMs). This clearly implies that unless the failure detection mechanism was unreliable or the strategy data changed, two modules cannot perform the same avionics function, and thus no two modules will reconfigure to the same lost function.

The discussion above shows that the reconfiguration strategy data will remain valid throughout the system lifespan, provided that the fault detection mechanism operates correctly, and no changes to the data are committed due to independent events such as hardware faults or natural phenomena.

## 8.5. Discussion on the failure detection mechanism

The failure detection mechanism is a key issue to the correct operation of the reconfiguration scheme. So far in the paper it was assumed that the mechanism operates without problems, and many proofs were based on such assumption. In this section the failure detection mechanism will be discussed, in order to analyse its reliability and limitations, as well as defining the required timing constraints.

The principles behind the design of the failure detection mechanism were discussed in detail in Chapter 5. These can be summarised as follows:

- each module monitors the backplane bus for transactions (messages) incoming from all other core LRMs and their avionics functions,
- messages are time stamped and recorded,
- if there is no backplane bus activity originating from an avionics function for a time longer than FAIL-DELAY specified for this function, it is considered lost,
- if the function is lost for the time longer than N * FAIL-DELAY, the (N-1)-th backup level module is expected to reconfigure.

In order to avoid single message upsets a function is considered lost only if K consecutive messages from the appropriate core LRM are lost.

> Commentary:
> In fact a function will be considered lost if the last recorder response is older than the function FAIL-DELAY. It is the length of the FAIL-DELAY time interval that will indicate how many messages will be used for failure detection, and this factor may vary depending on how fast particular functions generate messages.

The multiple messages based approach aims to eliminate problems related to random hardware faults of the backplane bus. It is clear that the mechanism reliability strictly depends on the factor K. On the other hand this is constrained by the time allowed for reconfiguration and the minimal time intervals between messages, i.e. should extremely large K be required, the time necessary for detection of a function loss could exceed the permissible bounds.

The analysis presented in Chapter 5 shows that such designed mechanisms are self-synchronising, and after a temporary loss of synchronisation they will regain correct operation provided that no persistent backplane bus errors are present in the cabinet (see Figure 5.1 in Chapter 5).

### 8.5.1. Reliability of the failure detection mechanism

The mechanism fault-tolerance depends on the relation between the time interval between consecutive messages produced by a function and the time required for reconfiguration a core LRM. Let $t_m$ denote the time interval between consecutive messages, $t_r$ denotes the time required for reconfiguration[77], K be as above, and $N_f$ denote the number of invalid messages that the algorithm can tolerate (i.e. the mechanism will operate correctly even though it will not have noticed $N_f$ first lost messages). The following equation shows the relation between the number of messages whose mis-detection can be tolerated and the other factors

eq. 86
$$t_r < (K - N_f) \times t_m$$
$$N_f < K - t_r/t_m$$

Commentary:
This can be easily obtained from the condition that reconfiguration of a lower backup level has to be completed before the second backup detects the need for reconfiguration, i.e.
$$t_m \times 2 \times K < t_m (K + N_f) + t_r$$

Equation eq. 86 shows that the shorter the reconfiguration delay, the higher the tolerance of the fault detection mechanism for given K and $t_m$. Clearly some realistic assumptions have to be made about the reconfiguration delay, as for $t_r$ equal to zero (timeless/immediate reconfiguration) the above equation would indicate that the mechanism can operate on a single message basis.

Commentary:
Note that $N_f$ describes the number of N first messages for which the function loss may not be detected and the mechanism will still operate successfully. For example, should $N_f$ be equal to three and the second message was the first mis-detected, the mechanism will only tolerate a mis-detection of the next message. This is illustrated by the following figure (Figure 8.1) , where the mechanism is able to tolerate a mis-

[77] This will include the period of time from detection of the need for reconfiguration to the moment when the first message from the function appears on the backplane bus.

detection of a single message, however two backup modules will reconfigure if the mis-detected message is not the first one and affects only the lower level backup.

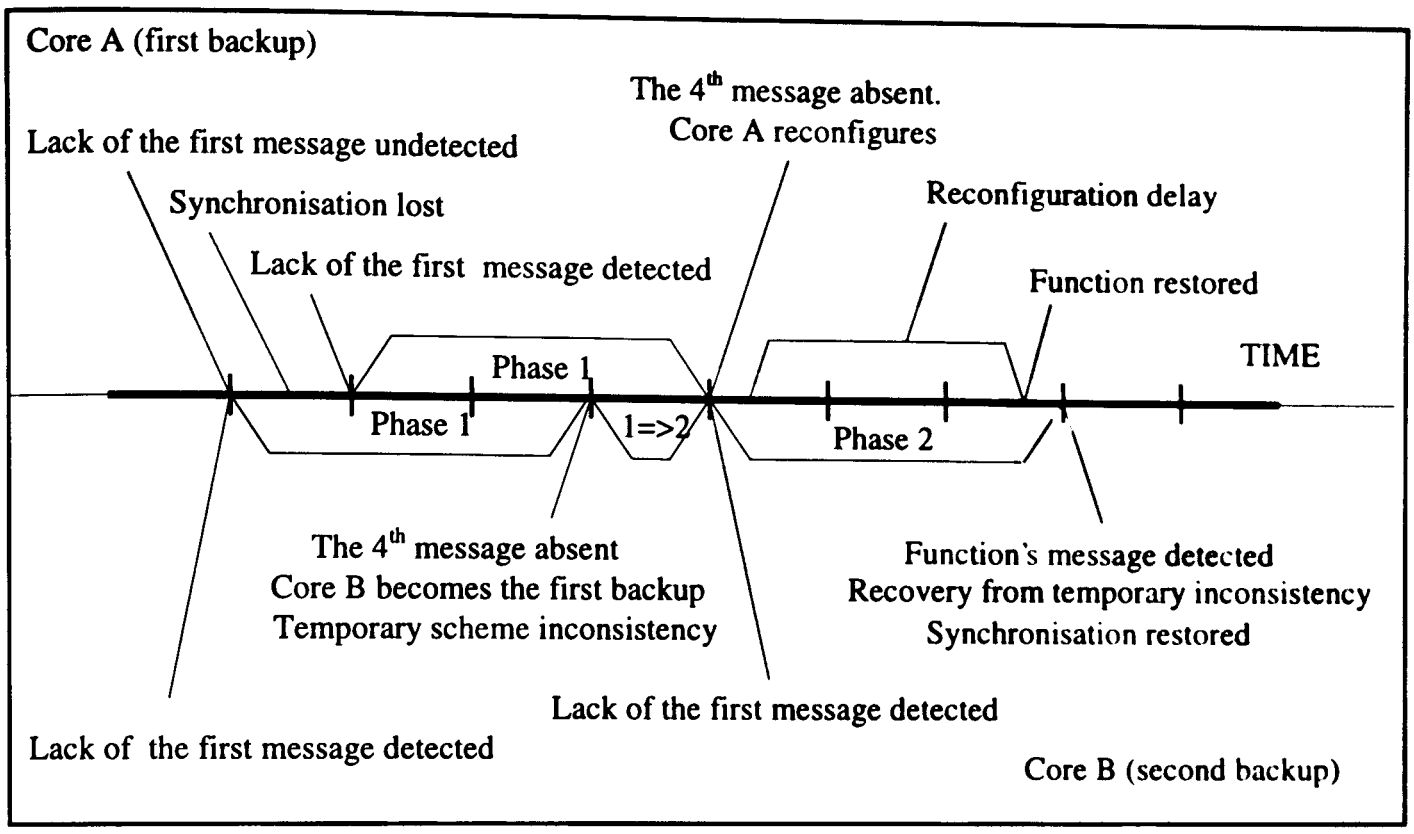



**Figure 8.1. Example of an invalid reconfiguration scenario due to backplane bus problem.**

Although the mechanism is able to tolerate some bus errors, the probability of postponed reconfiguration needs to be assessed. However, in order to be able to assess such probability the reliability of the backplane bus[78] would have to be known. As this is an unknown quantity for the future systems, the required reliability of the backplane bus (its Mean Time Before Failures - MTBF) will be found, on the basis that a critical function can only be lost with a probability lower than $10^{-9}$ per flight hour.

The configuration of a ten core LRMs cabinet will be considered for the purpose of this assessment. Each critical function requires four backup modules, thus it can only be lost if all of them have failed as well as the core LRM which performed the function originally. Following the analysis from Chapter 4 it will also be assumed that the aircraft can be dispatched with two modules in the cabinet already failed, each core LRM MTBF is equal to 20,000 hours, and the average duration flight lasts five hours.

[78] The notion of the backplane bus will be understood here as a single internally replicated communication media. Therefore, in practice the internal buses will be allowed to be of somewhat lower reliability.

In order to find the lower limit on the backplane bus reliability the following two cases need to be considered in the situation where an aircraft has been dispatched with two modules in the cabinet already off line:

- one of the backup modules fails during the flight, the critical function module fails subsequently and a backplane bus fault stops the remaining backup to reconfigure in time
- a critical function module fails, and a backplane bus error stops the remaining two backup modules to detect the failure in time.

A bus error may affect the failure detection mechanism if and only if it occurs shortly after the core LRM failure. For the purpose of this assessment the time interval of 0.5 second has been chosen (i.e. reconfiguration is expected to be completed within half a second), which in reality may be significantly shorter, what would provide additional tolerance for backplane bus errors.

Let $P_{core}$ denotes the probability per flight hour that a core LRM will fail, $P_{bus}$ denotes a similar probability for the backplane bus, dt denotes the time interval in which the bus failure must occur and T denotes the duration of the flight. The following equation denotes then the probability per flight hour that the first of the above described situations would happen

eq. 87
$$P_1 = (P_{core} \times T \times P_{core} \times T \times P_{bus} \times dt) / T$$

$$P_1 = P_{bus} \times P_{core}^2 \times T \times dt$$

From the above assumptions the following values will be substituted into equation eq. 87

$$P_{core} = 5 \times 10^{-5},\ dt = 1/7200,\ T = 5$$

As $P_1$ has to be lower than $10^{-9}$ $hr^{-1}$, the probability $P_{bus}$ can be obtained from the following equation

eq. 88
$$P_{bus} < 10^{-9} / ((5 \times 10^{-5})^2 \times 5 \times 1/7200)$$

$$P_{bus} < 576\ hr^{-1},\ \text{i.e. } P_{bus} < 0.16\ sec^{-1}$$

Thus the minimal required backplane bus MTBF is equal to 6.25 second, which should be easy to realise in practice.

In the second case the probability of occurrence per flight hour can be written as follows

eq. 89 $$P_2 = (P_{core} \times T \times P_{bus} \times dt)/T$$

$$P_2 = P_{core} \times P_{bus} \times dt$$

Commentary:
Strictly speaking the backplane bus failure would have to occur in a time interval shorter than dt, as the loss of the first N messages can be tolerated. However, the data bus MTBF calculated in this manner will allow for additional fault-tolerance, should the failure detection mechanism be able to operate properly despite occurrence of some backplane bus errors.

Substituting appropriate values into equation eq. 89, $P_2$ can be obtained as

eq. 90 $$P_2 = P_{bus} \times 1/7200 \times 5 \times 10^{-5}$$

As $P_2$ must again be lower than $10^{-9}$, the above equation can easily be solved with respect to $P_{bus}$ to see that

eq. 91 $$P_{bus} < 0.144\ hr^{-1}$$

Thus the minimal required backplane bus MTBF is now approximately 6.94 hours, which again should not be too difficult to achieve.

The analysis above shows, that provided that the replicated data bus MTBF is longer than 6.94 hours, the failure detection mechanism will be reliable enough not to permit a critical function loss with the probability higher than $10^{-9}$ per flight hour.

### 8.5.2. Probability of delayed reconfiguration

Having established the minimal required MTBF of the backplane bus, it is now possible to calculate the probability that at any stage of a five hour flight the reconfiguration will be delayed beyond the allowed time. For simplicity we will assume that reconfiguration will be delayed in a situation where a backplane bus error affects the failure detection mechanism within the 0.5 second time interval after the failure of a core LRM (the single remaining backup will not reconfigure in time). Thus such a probability per flight hour can be calculated from the following equation

eq. 92 $$P_r = (P_{core} \times T \times P_{bus} \times dt)/T$$

From the above equation $P_f$can be easily obtained as $10^{-9}$ $hr^{-1}$, and as such it will meet all the safety requirements.

Although the MTBF of approximately 6.94 hours is relatively long, note that not all backplane bus errors would affect the failure detection mechanism. Only those errors that alter appropriate messages in a specific manner (i.e. change the source of the message), are of interest at this point. Therefore, in practice bus errors may occur more frequently without an effect on the failure detection mechanism, and it is only required that this specific type of data invalidity must not occur more frequently than 6.94 hours.

### 8.5.3. Probability of detection of a non-existent failure

So far it has been established that in order to meet the safety requirements with respect to the loss of an avionics function for a time longer than the permissible limits, that backplane bus MTBF must not be shorter than 6.94 hours. In this section the probability that a backplane bus error will lead to invalid reconfiguration (i.e. reconfiguration without a failure) will be assessed. The following table (Table 8.1) shows the probability of detection of a non-existent failure for a failure detection mechanism based a varying number of lost messages (K)[79] and backplane bus MTBF of 6.94 hours.

| Number of messages (K) | Probability per flight hour |
|---|---|
| 2 | 1.00 E-10 |
| 3 | 2.96 E-16 |
| 4 | 6.25 E-22 |
| 5 | 1.02 E-27 |
| 6 | 1.37 E-33 |

**Table 8.1. Probability per flight hour of detection of a non-existent failure.**

For simplicity, in calculating the above figures it has been accepted that it is sufficient if a single bus error occurs in every K-th part of the time interval dt (0.5 second), to lead to invalid reconfiguration (see the equation below)

[79] This depends on the defined function FAIL_DELAY and the frequency with which the function messages appear on the backplane bus.

eq. 93 $$P_{mis\text{-}detection} = (P_{bus} \times T \times (P_{bus} \times dt/k)^{(k-1)}) / T$$

This assumes that the detection time for a single backup core LRM, i.e. the time when the failure detection mechanism is most susceptible to random bus errors, is very long and equal to dt. In practice, this would be shorter as both failure detection and reconfiguration must be completed within dt. Also, the number of messages in failure detection (K) can be much higher for some applications, see the following section, and thus it would provide additional safety margin. However, despite such strong and restrictive requirements the system shows that even for mechanisms based just on two messages (two consecutive messages must be lost to announce the function loss), the probability of detection of a non-existent failure is extremely low and will satisfy the safety requirements defined in [3] (the consequences of such mis-detection are discussed in section 8.5.5).

Another mode of failure which may lead to an invalid detection of a function loss relates to the problem of a lack of data bus communication for a period of time. In practice this would involve a lack of all messages related to a particular function for approximately 0.05 to 0.1 second (estimated detection time). However, as the loss of all backplane bus communications has to be considered catastrophic for the cabinet, the design of the data bus will have to take it into account for IMA systems, thus introduction of reconfiguration into the cabinet is expected not to require additional changes with respect to this matter.

### 8.5.4. Failure detection delays

There are two factors that influence the choice of the length of the failure detection phase - reliability of the detection mechanism (longer FAIL-DELAY and higher K), and the real-time constraints of the aircraft systems (shorter FAIL-DELAY and lower K). In order to provide highly reliable failure detection based solely on backplane bus monitoring, it must be based on multiple messages (see Chapter 5 for discussion). As shown in Table 8.1, the higher the number of consecutive messages that must be missing prior to detection of a lost function, the lower the probability that the failure detection mechanism will malfunction. Thus, in order to improve the mechanism reliability it would be beneficial to employ algorithms based on long sequences of messages. However, as discussed before, in some cases the function FAIL-DELAY may account only for a few messages if they are generated in long time intervals, and the need for the use of a greater number of messages would involve a change of the FAIL-DELAY interval for the function.

On the other hand, the system must detect the failure and perform reconfiguration within the strict time constraints, e.g. the loss of flight control systems should not occur for longer than some 400 ms in a commercial aircraft.[80] In static strategy data based reconfiguration schemes, this time interval must account for failure detection on all backup modules (up to four backup levels for the critical functions), and for single reconfiguration. Therefore, mechanisms based on long message chains (very long FAIL-DELAYs) may prove unfeasible.

It has been mentioned before, that it is important for the lower backup level to be able to reconfigure and resume the execution of a function before the next backup core LRM completes another failure detection cycle. Thus if $t_{fd}$ denotes the time required for failure detection and $t_r$ denotes the time required for reconfiguration the following relations must hold for each reconfiguration scheme:

eq. 94
$$t_{fd} + t_r < 2 \times t_{fd}$$

$$t_r < t_{fd}$$

To satisfy the above condition for the worst case scenario of the fourth backup reconfiguring within 400 ms, it becomes obvious that the total time allowed for reconfiguration, until the first message appears on the backplane bus, must be shorter than 80 ms.

> Commentary:
> If the time constraints cannot be met by the reconfiguration scheme for some of the "fast" avionics functions (whose loss can be tolerated only for a very short period of time), they may have to be replicated in the cabinet in a manner similar to traditional non-reconfigurable avionics systems. In this situation both copies would have to be presented in the strategy look-up table as separate functions of identical criticality. Thus the time required for reconfiguration can be eliminated in these cases at the expense of increased number of redundant processing modules.

Assuming that the time required for reconfiguration based on locally stored software and multiple APEX partitions (see section 8.6 for more discussion) should not exceed 80 ms, the failure detection mechanism should also complete within 80 ms. The time intervals between consecutive messages vary from function to function, and in general the more critical the function the more frequently its messages appear on the backplane bus. Recent talks with British Aerospace at Filton indicate that some of the critical functions

would send their messages as often as every five milliseconds, thus the number of messages used by the failure detection mechanism could be as high as twenty, which would provide extremely high reliability and it would still meet the required timing constraints.

In the case of less critical applications which communicate with longer time intervals, two solutions can be applied:

- the application can be modified in order to produce "empty" messages purely for the purpose of reconfiguration
- the FAIL-DELAY can be extended for those functions.

> Commentary:
> If an application produces its messages very infrequently, it will most probably be of low criticality and its loss can be tolerated for longer. Also, in the case of such functions the number of backup modules for the function will be much lower, and thus the required time interval will have to account only for failure detection on one or two core LRMs.

In the case of the first solution some problems can be encountered with the backplane bus throughput or access schedule, as additional messages will have to be broadcast. Also, additional effort would be required to identify and modify the applications software. Therefore, the second solution appears simpler, and it has been implemented by the reconfiguration scheme in the form of the variable *delays* (s.19.4) in the scheme state.

As it is impossible to define the failure detection delays in a more precise manner without a well defined set of system functions, these issues will not be further discussed in this chapter. However, in view of the above argument, it is expected that the failure detection mechanism can meet all the required timing constraints.

### 8.5.5. Limitations of the failure detection mechanism

The discussed mechanism provides reliable and flexible means for detecting failures of at least fail-passive processing modules, but its applicability is constrained by certain parameters critical to the

[80] Some authors envisage this time interval to be as low as 40 ms if military aircraft are also considered [51]

method. Those are predominantly related to the time constraints imposed by the aircraft systems (how quickly must the lost function be restored), and the hardware configuration and capacity of the processing units and the backplane bus (problems related to operation with simultaneous events are discussed separately in section 8.7.3).

The time in which the new function can be re-executed is constrained by the speed and capacity of the module hardware on one side, and by the system real-time requirements on the other. It is conceivable, that in some cases where a function must not be lost for longer than some 40 ms (e.g. flight control in an unstable military aircraft [51]), it will be difficult to detect a failure and to reconfigure a core LRM within 8 ms (immediate backup level). If faster and more capable hardware is unable to deal with the problem, it will not be possible to implement reconfiguration in such a class of avionics systems based on the principles discussed in Chapter 5 and Chapter 6[81].

The applicability of the discussed failure detection method is also constrained by the backplane bus reliability. As already discussed to meet the safety requirements the combined backplane bus MTBF must not be shorter than some 6.9 hours. This value has been calculated on somewhat severe assumption that any bus error encountered during the 0.5 seconds time interval after a failure will affect reconfiguration, as in practise it would have to alter a particular message and change its source in a very specific way. Therefore, a more realistic estimation of the backplane bus MTBF should take into account the number of possible values for the message source that could appear in the message. Assuming that all the values that could affect the reconfiguration process fall into the set of functions in the cabinet, the required combined backplane bus MTBF can be re-calculated in the following manner.

For the configuration of ten core LRMs in the cabinet, the source information can take one of the ten distinct values. A bus error would have to modify an existing message in such a way, that the message would seem to be coming from a failed module. Thus the following worst case scenario must be considered

- a core LRM performing a critical function fails

[81] Alternative techniques based on combining dedicated redundancy with in-flight reconfiguration could be sought for such systems.

- within 0.5 seconds of the failure, a bus error modifies the source of one of the messages
- the new source is identical to that of the failed module, i.e. there is a one to ten chance that the bus error would have caused reconfiguration to be delayed.

Equation eq. 92 becomes thus

eq. 95 $$P_r = (P_{core} \times T \times P_{bus} \times dt)/T \times 0.1$$

and the required MTBF can thus be reduced to some 0.69 hours. Note that both figures relate to the combined (replicated) backplane bus, and thus the MTBF of particular component buses could be slightly lower. Note also, that $P_r$ will in practice be lower as not all backplane bus traffic will be related to function messages (e.g. messages incoming from RDCs or smart actuators), which will further reduce the chance that a random bus error will affect reconfiguration, as messages from external sources are ignored by the reconfiguration scheme.

The discussion above shows that the reliability of the backplane bus should not constitute a problem (MTBF of 0.69 hours or even 6.9 hours should not be difficult to achieve), when applicability of the failure detection mechanism is being considered. Therefore the main limitation of the mechanism relates to the restrictions of the length of the FAIL-DELAY and the number of messages (K) used during failure detection.

## 8.6. Time constraints conformance

In this section the real-time aspects of avionics systems implementing reconfiguration schemes will be discussed. As both the aircraft specific time constraints and the capacity of the hardware are unknown at this point, the discussion will focus on identifying the real-time issues and classes of avionics systems to which the reconfiguration scheme can be applied.

Three main aspects of the reconfiguration scheme have to be investigated when the real-time operation of a reconfigurable avionics system is being considered:

- fault detection delays
- reconfiguration delays

- issues related to concurrent or simultaneous execution of the reconfiguration software and the avionics application.

Issues related to fault detection delays have already been discussed in sections 8.5.4 and 8.5.5 of this chapter, thus, they will not be further investigated in the remainder of this section.

### 8.6.1. Reconfiguration delays

As previously mentioned, the time in which the lost function must be re-executed may be as short as 400 ms. It was also discussed in section 8.5.4 that such time constraints require the actual change of the module function to be completed within some 80 ms. The following actions need to be performed when a new applications is to be executed by a core LRM:

- termination of the current function
- downloading of the required application software
- identification and downloading of the necessary state information
- execution of the new function.

The first activity should be achievable in a straightforward manner with the use of some OS services such as de-allocation of the time window or reclamation of resources, and should not lead to significant delays. It is expected that less than one millisecond should be required to render a process inactive in the environment of a processing module.

The way in which the module handles software downloading is expected to be the deciding factor. As discussed in Chapter 5, it is unlikely that downloading the software from an application module via a dedicated bus will be able to meet all the timing constraints. Simply the throughput of the software bus would have to be of the order of 100 Mbit/sec to allow downloading of 1 MB of data within 80 ms not leaving any time for context switching or the state operations. Therefore, it is expected that the software for any application that the core LRM may ever be expected to perform will reside in the module non-volatile memory. It is conceivable that the memory to memory transfer can significantly exceed the required rate of 100 Mbit/sec, and thus it will be able to meet all the timing constraints.

Low prices of RAM suggest that all the required software could be loaded on each core LRM during system initialisation, and thus no additional downloading will be necessary on reconfiguration. Non-volatile memory would than be used only to provide backup in case of a power loss, and to allow uploading of the software during the system start-up. This solution should be particularly beneficial in systems implementing ARINC 653 APEX [10], that supports multiple partitions on a single processing unit, some of which may be inactive. If all applications already reside in the core LRM RAM, the delays related to software downloading can be eliminated and the cost of a termination of an avionics function and of activation of a new partition can be minimised. In this case only a change of the partition flag in the APEX environment would be required to render the application active or inactive.

The time required for identification and fetching of the application most recent state can also be very short. In the simplest implementation an APEX service could provide an address in the memory where the application state is stored. The relevant avionics function can than handle the data in any desired way, while the reconfiguration software does not require any knowledge of the state parameters, and does not have to understand the information. The application state is expected to be small (possibly containing only values for a few required parameters), thus state retrieval activities should not impose notable delays.

In view of the above argument it can be stated that the aforementioned reconfiguration schemes should be able to meet the timing constraints for commercial aircraft, and should also be potentially applicable to more demanding systems, particularly if the software downloading phase is eliminated due to employment of multiple inactive partitions.

### 8.6.2. Simultaneous execution of reconfiguration software and an avionics application

The reconfiguration scheme services can be provided either via a dedicated process running on the same module as the avionics application, or they can be embedded into the core module APEX. In either case some additional processing power will be required in order to allow dynamic reconfiguration of the system. In this section the complexity of the reconfiguration routines and the time required for their execution will be assessed. As the targeted hardware and software platforms are still mostly undefined, the assessment will be based on a simulated system, both hardware and software-wise.

Low complexity of the reconfiguration software (see Table 8.2 for details) suggests that its processing power requirements will be very low. In order to assess the CPU time requirements related to various aspects of the reconfiguration scheme, a simulation of the reconfiguration services has been implemented.

| Reconfiguration software activity | Lines of code | No. of calls to sub-routines[82] | No. of executed lines |
|---|---|---|---|
| receiving messages[83] | 4+6+3 | 2 | 13 |
| analysing and recording messages[84] | 8 | 0 | 8 |
| check of the reconfiguration criteria | 2+7+2 | 3 | 13 |
| partition activation | 12 | 1 | 13 |
| termination of current function | 7 | 1 | 8 |
| total | 51 | 9 | 55 |

**Table 8.2. Complexity of the reconfiguration software based on implemented simulation.**

A great care has been taken, to make the reconfiguration software as representative as possible for the purpose of this analysis, however due some hardware limitations certain aspects of the targeted system (e.g. backplane bus access) could not have been closely simulated. The following table gives a brief comparison of the expected final system and the simulated one.

[82] Not including the implementation of system services.

[83] Numbers based on a backplane bus implementation simulated via UNIX sockets..

[84] This does not include the implementation of the system clock provided by the OS.

| Aspect | Simulated configuration | Targeted system |
|---|---|---|
| Operating system | MS-DOS 6.22 | unknown OS + APEX |
| Processor | Pentium™ 133 MHz | unknown |
| Backplane bus access | simulated via memory access | probably via access to bus registers or buffer |
| Timer | standard clock | unknown |
| Function termination | simulated via a system call | probably based on partition flags |
| Function initialisation | simulated via the "execlp" system call to execute a short[85] program | probably based on partition flags |
| State | memory access | memory access |

**Table 8.3. Comparison of an expected final system and a simulated one.**

The simulation has been carried for the worst case scenario with respect to the processing power requirements, i.e. each function of the reconfiguration software has been called in each cycle. This corresponds to the situation where the only module operating in the cabinet is the most critical function core LRM, whose messages are the only ones present on the backplane bus (messages related to the external sources are irrelevant). In such a situation the processing module continuously detects the loss of other functions, and the reconfiguration conditions are checked for each function with every arriving message. The software has been modified for the purpose of simulation to enforce the check of all the reconfiguration conditions (including the strategy table check) with every call to the reconfiguration routine. In a real system the processing power requirements would be somewhat lower, as the function criticality check would be sufficient to reject reconfiguration in many cases. The results of the simulation are shown in the following table.

| Number of simulated cycles | Time required for computation |
|---|---|
| 100 | 160 ms |
| 500 | 160 - 220 ms |
| 1000 | 220 ms |
| 10,000 | 1.1 - 1.6 sec |
| 50,000 | 5.16 - 5.22 sec |
| 100,000 | 10.27 sec |

**Table 8.4. Simulation results I.**

[85] Due to the single tasking environment of MS-DOS, the application was designed to finish almost instantaneously

As MS-DOS is a single threading operating system there has been some overhead included in the results related to the set-up phase, where the timing program had to execute the simulated application. This can be easily extracted from the results for a very small number of messages which are virtually identical, i.e. the overhead was greater than the time required for computation. From the results obtained for a great number of simulated messages it is clear that the time required to process a single message in a worst case scenario with no reconfiguration is as short as 0.1 ms. Moreover, with the aid of profiling tools it has been found that some 82% of the execution time was related to time reading (inefficient MS-DOS system call), therefore the implementation of an optimised timing device could lead to a further reduction of the processing power and the time requirements.

The simulation software has been subsequently modified to investigate the case where the processing module had to reconfigure. Again the worst case scenario has been chosen for the simulation, where a core LRM checks all the reconfiguration conditions for all the functions in the cabinet with every message, and it reconfigures once in each cycle. The simulation results are shown in the following table.

| Number of simulated cycles | Time required for computation |
|---|---|
| 10 | 2.90 sec |
| 50 | 13.9 sec |
| 100 | 27.69 sec |

**Table 8.5. Simulation results II.**

In the case where reconfiguration was required in each cycle (i.e. with each received message), the time required for completion of a cycle has been estimated around 277 ms. Such long time delays could not be accepted by the targeted avionics system. However, as mentioned before the simulation has been conducted on a single threaded OS, where process management has been simulated via system services, and thus it was far from optimal. This has been confirmed by the profiling tools, which showed that approximately 98% of the CPU time was used to start and terminate processes.

In an attempt to optimise the task of process management, the same software has been adapted for execution in a multitasking environment (a "fork" system call has been performed in an environment of IRIX 6.2), and has been subsequently run on a Silicon Graphics server. The simulation has shown that the CPU time required for execution of the full cycle was not longer than some 34 ms. The accuracy

limitation of the timing software did not allow more precise specification of the time, and the results encompass

- four milliseconds for main application cycle execution
- approximately ten milliseconds for a call to system routine "fork"
- approximately ten milliseconds for the shell call used to terminate the current application ("kill" routine)
- approximately ten milliseconds for starting the new application.

This is believed to be acceptable in RIMA systems.

In conclusion it can be said that the simulation results have verified the feasibility of simultaneous execution of the reconfiguration software and an avionics application within the time constraints imposed by the real-time aspect of RIMA systems, provided that optimised OS/APEX services for time reading and process management will be available.

## 8.7. Failures of the reconfiguration scheme

As discussed before, the reconfiguration process operates in three phases: failure detection, check of the reconfiguration conditions and finally re-activation of the desired application. Each of these phases is potentially susceptible to hardware/software problems, that in turn could lead to invalid operation of the scheme or even to its failure. As issues related to invalid failure detection have already been discussed in the previous section, the discussion here will focus on the two remaining phases - check of the reconfiguration conditions and the actual reconfiguration of an avionics function.

### 8.7.1. Erroneous check of the reconfiguration conditions

In this section the following two cases will be considered:

- the loss of an avionics function has been detected but due to some internal problem the core LRM fails to reconfigure, despite being assigned as a backup for the function
- the loss of an avionics function has been detected and multiple modules reconfigure to sustain execution of said application

As for the purpose of this discussion the failure detection mechanism is assumed to operate without errors, either of the situations can only occur if the reconfiguration strategy data stored by the core LRM has been modified. It has been discussed in section 8.4 that the initial strategy table is consistent with system requirements and the data remains valid throughout the system lifetime. However, some external hazards such as neutrino bombardment could lead to changes of the contents of particular memory locations, which in turn could invalidate the data. Thus the probability of such changes and their consequences have to be considered.

The second situation seems to be somewhat less critical than the first one, however, the long term effects can be similar in both cases. If two core LRMs reconfigure to the same function, a less critical function will be lost but the system will incur no other immediate degradation, provided that smart actuators can operate properly with commands arriving from multiple sources (some adjudication mechanism may have to be required). However, after invalid reconfiguration the module cannot be considered as an autonomous backup for any application (as discussed before both modules will act identically on received messages), therefore should the module be required to provide backup for an avionics function, the system operation will be degraded. Such an effect is similar to the first failure scenario, where invalidity of the reconfiguration data prevents a module from reconfiguring (i.e. the module does not provide a backup for a function).

A change of a single entry in the strategy table can lead to either of the discussed situations, and thus such an event must be at least as improbable as the loss of the module itself. It is estimated that the phenomenon of neutrino bombardment affecting a memory location may happen as often as once in 3000 hours. However, it will affect at random any of the memory locations and thus the probability per flight hour of the strategy table being affected is much lower. For example in a core LRM with 8 MB of RAM and the strategy table size of 1 kB, the probability that the phenomenon of neutrino bombardment would affect the table is as low as $2.46 \times 10^{-7}$ $hr^{-1}$. This is lower than the probability of losing the whole core LRM, and as such does not increase the safety hazard. Moreover, the static strategy table could conceivably be placed in a read only memory (ROM), which would protect the data from neutrino bombardment and similar phenomena.

### 8.7.2. Failure to re-execute an application

There are two main problems related to restarting the application address the issue of validity of the application software and the function state. The phenomenon of neutrino bombardment can affect the application software and its recently saved state in the same manner it can affect the strategy table. However, particularly in the case of the software, the chances of such an occurrence are much higher as the program itself is many times bigger than the strategy look-up table.

Clearly some of the memory locations can be changed within the software without any adverse effects, others may cause only minor problems (e.g. changes to non-critical parameters or variables). Therefore it is extremely difficult (if at all possible) to find an exact probability for critical alteration of the software due to natural hazards. In order to eliminate such phenomena each core LRM could store multiple copies of the software and check their integrity (this would lead to rather high memory requirements), or the system could utilise the idea of corrective codes to identify and eliminate software errors. This would increase the complexity of the system and could induce some processing overhead in the system, but would reduce the risk related to natural hazards.

> Commentary:
> In the case of MS-DOS based PC hard disk storage devices, problems related to unwanted changes of the memory locations are normally dealt with on the hardware level with the use of the Error Correction Code (ECC) [52]. It is possible that the implementation of software store on each of the core LRMs could use such a code to detect and correct spontaneously appearing errors prior to the use of the software. [86]

Should the software be detected invalid beyond the corrective capacity of the code, the module will not be able to reconfigure to the function, and the next backup (if available) will have to do so. However, it is unlikely that the software for other functions will also be affected, and thus the core LRM should be allowed to operate and provide backup for other applications.

The problem of state invalidity raises additional issues when re-execution of an avionics application from a saved checkpoint is being considered. In the case where the state information has been affected and such an event has not been detected, the application may restart from an invalid checkpoint producing

erroneous results. This situation could lead to a latent failure, where a core LRM would perform a function producing undesirable but consistent results, and thus multiple processing channels would not be able to detect the problem. In long term operation such behaviour could manifest itself with discrepancies between the calculated values and the sensor reading.

Although the invalidity of the state can lead to a potentially hazardous or catastrophic behaviour, the problem is relatively easy to deal with. It is expected that the state information will be much smaller than the application software, and thus it will be easier to store and compare multiple copies of each saved state. Also, the probability of a natural phenomenon affecting a small amount of data is much lower than the probability of losing the whole core LRM due to a random hardware fault (see section 8.7.1).

### 8.7.3. Operation with simultaneous events

The probability of two failures occurring nearly simultaneously during an average duration flight can be easily calculated from the following equation

eq. 96 $$P_{sim} = P_{core} \times T \times P_{core} \times T \times dt / T$$

where $P_{sim}$ is the probability per flight hour of two processing modules failing almost simultaneously, $P_{core}$ is the per flight hour probability of a core LRM failure ($5\times10^{-5}$ $hr^{-1}$), T is the flight duration (five hours) and dt is the constraining time interval (here again it has been assumed to last 0.5 second). Substituting the required values in to equation eq. 96 one can read sought probability as

eq. 97 $$P_{sim} = 0.174 \times 10^{-9}\ hr^{-1}$$

which shows that the event of two core LRMs failing nearly simultaneously is extremely improbable.

The probability that two modules will recover simultaneously, or that a module will recover almost at the time of another core LRM failure is more difficult to calculate as the probability of module recovery per flight hour is an unknown quantity, and will have to be assessed separately when appropriate data is available.

---

[86] Various error detection and error correction codes can be found in [53], whilst another example of fault-tolerant disk storage is presented in [54].

Commentary:
It is expected that it can be similarly low as $P_{sim}$, as one has to look at the combined probability of a failure and recovery of a core LRM (it must fail prior to recovery) and another failure that must occur nearly simultaneously, or two failures and subsequent simultaneous recoveries. It is possible that the system might have been dispatched with a failed module, however if the module subsequently recovered, it is clear that it was a temporary failure which most probably would have been eliminated between flights and would not have affected the probability.

The only situation where multiple modules recover at the same time occurs during system initialisation, and it is guaranteed to be deterministic and correct by properties P7 and P8.

Two simultaneous events can affect the reconfiguration scheme in the following ways:

- scheme will not behave deterministically and possibly with longer reconfiguration chains
- two modules will reconfigure to the same function.

In the first situation at least two processing modules must fail nearly simultaneously in order to render it impossible to guarantee the order of detection of the failures. The same situation relates to cases where two modules recover at the same time, or one module fails and a different one recovers almost simultaneously. Similarly as in the first situation it is impossible to guarantee the order to detection of the failures, which will depend on the timing of each event.

In general, the determinism of the scheme will have to be questioned should the probability of occurrence of simultaneous events be higher than the extremely improbable level defined in [3] as $10^{-9}$ $hr^{-1}$.

## 8.8. Conclusions

It has been shown that that reconfiguration schemes based on presented specification can meet the safety-critical related requirements, and they can operate within strict real-time constraints of avionics system in civil aircraft. It has been also confirmed that the scheme will operate deterministically provided that no nearly simultaneous events of failure or recovery will occur in the system.

The presented concise VDM specification of the method allows relatively easy implementation of the reconfiguration scheme on any desired platform. This is planned as the next phase of the research which

will be completed on the hardware representation of an IMA system provided by British Aerospace and possibly on another alternative system. This is expected to allow for identification of practical issues related to implementation of dynamic reconfiguration into avionics systems, which in conjunction with this chapter should constitute a good basis for identification of the related certification issues.

# Chapter 9. Practical Implementation of a Dynamically Reconfigurable Autonomous System

## 9.1. Introduction

A number of possible reconfiguration schemes suitable for Reconfigurable Integrated Modular Avionics (RIMA) have been identified and described in Chapter 6. A reconfiguration scheme appropriate for RIMA has been devised, formally described and discussed in Chapter 7 and Chapter 8. Following the completed work, the implementation phase was to identify various practical issues that need to be taken into account while constructing a RIMA system.

Although it was clear that the implementation will not be thoroughly representative as neither was a completely adequate hardware platform available, nor was the APEX/OS environment fully conforming with the ARINC 653 specification [10]. However, despite the inevitable deficiencies, it was hoped that the working implementation of a reconfigurable system will provide an interesting insight into different aspects of dynamically reconfigurable systems.

The work focused on three main areas of

- inter-module communications
- process/task management
- real-time operation.

Two different hardware and OS platforms were used to implement the system, allowing a comparison of various features and their impact on the system performance to be made.

The remainder of this chapter describes both systems used for implementation (section 9.2), focuses on certain aspects of the reconfiguration software principles and coding (section 9.3), and then discusses various practical issues that were identified during the implementation and testing of the RIMA system as potentially problematic (section 9.4).

## 9.2. System description

The demonstration of dynamic reconfiguration of avionics functions has been conducted on two systems based on RIMA architecture "C" as proposed in Chapter 2. One implementation was conducted in the Systems Digital Control Laboratory (SDCL), courtesy of British Aerospace AIRBUS Ltd. at Filton, UK, whereas the other was based on a more open system and was conducted at the University of Bristol, England.

Due to cabinet size limitations it has been decided that no more than five core LRMs will be used to conduct the demonstration, and one or two gateway modules will be employed to represent data exchange with external devices. As the reconfiguration scheme can be easily scaled to fit a wide range of cabinet sizes, this has been assumed to be sufficient.

Although the ARINC 651 document [2] proposes the use of ARINC 659 standard [9] based data bus for backplane communications, due to the unavailability of such a data bus resulting from its extremely high cost, alternative asynchronous data buses have been used in both implementation. This has allowed the identification of a range of issues related to the operation of the failure detection mechanism with asynchronous communication media, that are discussed later in this chapter.

The modules used in the implementation were not initially fail-passive, and this function has been provided by a human controlled fault induction mechanism, ensuring that all module activity will be terminated on failure.

### 9.2.1. SDCL-based system

Courtesy of the British Aerospace personnel at Filton, various practical implications related to implementation of reconfigurable avionics were first investigated with the use of the SDCL. The system, originally used as an implementation of an IMA system, comprised a number of core modules placed in a cabinet and communicating with one another on the backplane bus. Moreover, the original system also used multiple gateway modules to interface the ARINC 629 data bus with peripheral devices (actuators,

landing gear, etc.) However, at the stage when RIMA was to be implemented on the SDCL, the external devices were no longer available as the system was undergoing some major architectural changes.

Thus, the system used for implementation of RIMA included up to four core LRMs and a single gateway module set to emulate communications with the external devices, although no actual data was sent onto the ARINC 629 global system data bus. The processing modules and the gateway were placed in a cabinet and were connected via an FDDI ring serving as a backplane bus.

#### 9.2.1.1. Modules

Each of the processing modules was based on a Motorola CPU (M68040) and had 4 MB of Random Access Memory (RAM). Each module was connected to the backplane bus (FDDI ring) as well as Ethernet network, although only the former has been used for communication between the applications and the reconfiguration processes (see section 9.2.1.3).

#### 9.2.1.2. Operating system

Each core LRM was running a VxWorks Tornado operating system [55] based to some extent on UNIX principles. The system allowed multiple processes to be scheduled for execution on a single CPU, however, it did not provide memory partitioning, as any task could access the memory space of other tasks. The possibility of such an undesirable interaction of processes precludes the use of such an operating system in commercial aircraft.

The OS does provide other features that make it attractive for implementation of RIMA. These include:

- socket layer interface to the data bus (both BSD and Posix standards), allowing a simple and portable code to be written when sending and receiving messages from the communication media (see section 9.3.4),
- a system timer incremented with each tick of an internal clock whose frequency can be set as required (60 Hz have been used for implementation, see section 9.3.6),
- easy to use (although not portable), system calls allowing for termination and activation of tasks [56], [57].

#### 9.2.1.3. Data bus

The data bus used for backplane data exchange was a fibre optic 144 Mbit/sec FDDI ring, where messages sent onto the data bus are passed from one core LRM to another, until they are finally removed upon their return to the sender. In the case of broadcast messages, this implies that different modules will receive the message at different times depending on their position in the ring.

To complicate the implementation, the FDDI ring operates asynchronously and the messages sent by particular processing modules can be buffered at the core LRM pending the data bus access. This poses certain hazards when timing of handshake or data messages is being considered. In general, it is impossible to guarantee when the message will arrive on the bus and in what order the messages will be sent and received on different cores. Although, in principle an upper bound of the time delay could be found, as this depends on various factors such as the current system load and the amount of the data bus traffic, the actual delay would be different for different systems.[87]

Finally, the VxWorks orientated software driver for the FDDI bus was a large and unsupported program, which further complicated the analysis of the timings and the actual behaviour of the data bus traffic.

Practical aspects of the use of an asynchronous FDDI ring for backplane communications are further discussed in section 9.4.2 of this chapter.

### 9.2.2. UNIX configuration

The second implementation of a reconfigurable avionics system was conducted on an open system at the University of Bristol. The University configuration comprised a number of UNIX workstations (Silicon Graphics and PC) interconnected via an asynchronous 10 Mbit/sec Ethernet data bus (a Local Area Network, LAN). The workstations were mainly used to act as the processing modules, although some of them were also used to work as gateways and to emulate the external traffic. Up to five core LRMs and a single gateway module were implemented and subsequently tested on the UNIX configuration.

---

[87] In [63] the author attempts to construct a real-time communication medium based on an asynchronous LAN, however, a great deal of information about expected data traffic, processor utilisation etc. is required for the assessment of various communication delays.

Unlike in the case of the SDCL, the system was not purely dedicated to RIMA and other users were permitted to interact with the data bus and the workstations at the time the reconfigurable system was in operation. This in principle led to an increased communication load on the Ethernet network, greater requirements for processing power and variable scheduling of particular tasks. To minimise such undesirable interferences, the system was tested when the independent data bus traffic was minimal, and the workstations used to run as core LRMs and gateway were lightly loaded.

#### 9.2.1.1. Modules

As mentioned before, two different types of machines were used to implement processing and gateway modules. The majority of the machines were Silicon Graphics (SGI) Indy 2 workstations whose configuration is shown in Table 9.1 below.

| Iris Audio Processor: version A2 revision 4.1.0 |
|---|
| 1 133 MHz IP22 Processor |
| FPU: MIPS R4600 Floating Point Coprocessor Revision: 2.0 |
| CPU: MIPS R4600 Processor Chip Revision: 2.0 |
| On-board serial ports: 2 |
| On-board bi-directional parallel port |
| Data cache size: 16 Kbytes |
| Instruction cache size: 16 Kbytes |
| Main memory size: 96 Mbytes |
| Vino video: unit 0, revision 0, IndyCam not connected |
| Integral ISDN: Basic Rate Interface unit 0, revision 1.0 |
| Integral Ethernet: ec0, version 1 |
| Integral SCSI controller 0: Version WD33C93B, revision D |
| Disk drive / removable media: unit 2 on SCSI controller 0 |
| Disk drive: unit 1 on SCSI controller 0 |
| Graphics board: Indy 8-bit |

**Table 9.1. Configuration of the SGI workstations (output of hinv command).**

It should be emphasised that the drive on which the software for the reconfiguration process and the avionics applications was stored was NFS mounted from an external file server. This influenced to some extent the timing of reconfiguration, as the software had to be fetched every time the application was to

be spawned. However, as this related to small programs and was handled over a fast LAN, associated delays have been proven to be very short (see section 9.4.3).

The second type of processing unit used for the simulation was an Intel processor based Personal Computer (PC), whose configuration is shown in Table 9.2 below.

| Processor: Pentium™ 133 MHz CPU |
|---|
| Main memory size: 64 MB |
| Secondary cache memory size: 256 kB burst cache |
| Video card: 2MB STB PowerGraph 64 Video (Trio64V+) |
| Network interface: 16 bit 3COM Ethernet card (3c509) |
| Disk drive: Western Digital 2 GB IDE drive, WDC AC32100H |

**Table 9.2. Configuration of the PC workstation.**

#### 9.2.1.2. Operating system

In both cases, the SGI machines and the PC, the operating system used was UNIX. The SGIs ran the Silicon Graphic proprietary IRIX 6.2 software, and the PC ran a freely available Linux release from Redhat (version 4.2).

As mentioned above, both systems are UNIX clones and therefore support multiple processes sharing one or more CPUs. Unlike the VxWorks system though, they maintain some memory partitioning between tasks, thus minimising the risk of data or process corruption. In terms of data bus connectivity, both systems provide a socket layer interface complying with both BSD and Posix standards.

#### 9.2.1.3. Data bus

The data bus used in the UNIX configuration was a 10 Mbit/sec Ethernet network. Similarly to the FDDI bus used in the SDCL configuration, the Ethernet based data bus operated asynchronously, and thus messages were queued at core LRMs awaiting for the bus access. In this situation it was again impossible to guarantee the timings and the order of messages being sent by particular processing modules.

Also, as the core modules were scattered across the building rather than placed in a single cabinet, the physical distances the message had to travel were much longer than in the case of SDCL. However, this was a negligible factor when the speed of light at which the signal propagates was taken into account

## 9.3. Implementation

The reconfiguration scheme whose implementation is discussed in this section is based on the formal specification as described in Chapter 7.

An independent reconfiguration process runs concurrently with the avionics application and monitors the backplane bus to identify the need for reconfiguration. Should reconfiguration be required, the reconfiguration process has the authority to terminate the current function and to start the new desired application. The reconfiguration process monitors the backplane bus for handshake messages arriving from all core LRMs, as well as keeping track of the most recently received states of the avionics applications. The state information is subsequently made available to the application during reconfiguration. As the reconfiguration process has the capacity to terminate any application running on the core, the integrity of its implementation should be considered as a safety-critical issue.

### 9.3.1. Reconfiguration scheme

The function based static strategy table approach as discussed in Chapter 7 and Chapter 8 has been chosen for implementation of the reconfiguration scheme on both the SDCL and UNIX platforms. The choice followed the fact that such reconfiguration schemes are most promising for RIMA systems due to their simplicity and generically embedded deterministic functional system degradation (see Chapter 6 for discussion).

The strategy table used to govern reconfiguration comprised five core LRMs, each running an avionics function of different criticality as shown in the following Table 9.3.

| | Catastrophic | Hazardous | Major | Minor | Redundant |
|---|---|---|---|---|---|
| Catastrophic | -1 | -1 | -1 | -1 | -1 |
| Hazardous | 3 | -1 | -1 | -1 | -1 |
| Major | 2 | 2 | -1 | -1 | -1 |
| Minor | 1 | 1 | 1 | -1 | -1 |
| Redundant | 0 | 0 | 0 | 0 | -1 |

**Table 9.3. Strategy table used in the implementation of the reconfiguration scheme.**

The use of the strategy table and the failure detection mechanism were discussed in detail in Chapter 5 and Chapter 8.

### 9.3.2. Applications

The actual avionics applications used by the SDCL were unavailable for this project, and thus separate programs have been written to emulate the avionics functions. As the external devices were no longer connected to the cabinet, it was considered sufficient if the applications produce their messages in given time intervals, and the actual contents of the data was deemed irrelevant.

Therefore the applications focused on producing handshake, data and state messages in required time intervals (see Table 9.4 below) to emulate the avionics functions and to allow investigation of the feasibility of state propagation and reconfiguration based on backplane bus monitoring.

| | Intervals (ticks) | | Msg count per sec (approx.)[88] | |
|---|---|---|---|---|
| Function | Handshake | State | SDCL | UNIX |
| Application 0 | 1 | 10 | 126 | 140 |
| Application 1 | 2 | 20 | 63 | 70 |
| Application 2 | 6 | 42 | 21 | 24 |
| Application 3 | 6 | 42 | 21 | 24 |
| Application 4 | 6 | 42 | 21 | 24 |
| **Total**[89] | | | **253** | **281** |

**Table 9.4. Message timings and message count for particular applications.**

The handshake messages were only six bytes long consisting of a four byte header, one byte identifying the type of the message, and one byte indicating its source (the ID of the avionics application that produced the message). The state messages were slightly longer, as in addition to the above discussed six bytes they also contained simple state information (application message counter), which allowed activation of avionics functions from a state different than initial one.

---

[88] Including data messages for external devices that were sent in identical time intervals as the handshake messages

[89] Note that the total number of packets takes into account fractional numbers, that have been rounded to integer values in other table entries.

Commentary:
Only the response and application state messages were formulated according to the above protocol as the reconfiguration process monitored only these messages. Data messages were directly sent to the gateway module and were not received by the reconfiguration software. They were not broadcast, and they were sent to different communication ports to avoid conflicts.

Such an implementation of pseudo-avionics applications allowed completion of this phase of the research within the time constraints, and yet allowed for representative observation of the system behaviour with respect to autonomous dynamic reconfiguration. However, it did not investigate in depth issues related to the notion of state in complex applications, its determination and propagation that would arise, should the actual avionics functions be required to start from a state different than the initial state. These issues have been discussed for example in [18], [25] and [23], and should be subject to subsequent research with respect to RIMA (see Chapter 11).

### 9.3.3. Gateways

Similarly as in the case of the avionics functions, the gateway code originally used in SDCL was not available, and it would be of little interest as the external devices were no longer connected to the cabinet. Given this situation, some new gateway code was written to receive data messages produced by avionics functions and to emulate data packets arriving from outside the cabinet (the gateway code generated such messages and passed them to the applications).

It was assumed that the applications receive feedback from the peripheral devices less frequently than they produce their data, therefore the gateway software has been tuned to generate quasi-external messages roughly every tenth message received from the applications (see Code example 9.1 below). Such implementation allowed for a representative observation of the backplane bus traffic and its influence on the system behaviour, and as the pseudo-avionics applications only emulated the actual functions the contents of the "external" packets generated by the gateway software were irrelevant. As the gateway modules were not connected to the system data bus (see discussion in section 9.2.1), the data messages received from core LRMs were discarded (it was physically impossible to forward them to any external devices).

```
/* Gateway operates in an endless loop */
while(1){
   int msg_len;                    /* Message length indicator */
   char data[MAX_MSG_LEN];         /* Buffer for message data */
   struct sockaddr_in address; /* IP address from which the
                               message was received */

   /* If get_message return non-zero, a message was received from
   the backplane bus from a module whose IP address has been
   recorded in the variable 'address' */
   if ((msg_len = get_message(data, input_socket, &address))!=0)
   {
      /* Send simulated forwarding external messages, but only on a
      random 1 to 10 chance (this can be easily re-tuned to a
   different factor) */
      if (random()%10 == 0){
         strcpy(data,"external data");
         send_message(data,output_socket, &address,
                        CORE_INPUT_PORT);
      }
   }
}
```

**Code example 9.1. Simulated gateway - main loop of the workstation code.**

One could argue that such implementation of the gateway software did not fully investigate the CPU requirements for the gateway modules. However, it has been observed that a Silicon Graphics workstation (as described in section 2.1.1 of this report) running the most CPU intensive pseudo-avionics application (the most critical applications produce their messages most frequently and thus require most CPU time), the reconfiguration process and the gateway process together had only used some 70% of its CPU power. This was apportioned as follows:

- the gateway process used at most 7% of the CPU time,
- the reconfiguration process used up to 8.8% of the CPU time,
- the application used up to a massive 54.6% of the workstation CPU time.

To make these results representative, three additional applications and reconfiguration processes were running on separate workstations to emulate the complete working system.

These observations indicate that the CPU requirements of gateway modules should not be difficult to satisfy even with more complex implementations allowing interaction with the external devices.

### 9.3.4. Inter-module communications

In both implementations (SDCL- and UNIX-based), the socket level interface was used to implement communications on the backplane bus. The User Datagram Protocol (UDP) was used, as it allows for data packets to be dropped by the system, and thus it simulates random bus errors.

> Commentary:
> UDP does not guarantee delivery of the messages, and depending on the state of the connection between the communicating hosts it may lose some packets (it is considered as an unreliable datagram protocol).

In order to allow the reconfiguration processes running on all core LRMs to receive the handshake and state messages from each application, the relevant packets were broadcast on the data bus to make them available for all connected hosts. The use of the broadcast facility reduced the number of packets present on the backplane bus at any time (one packet was broadcast instead of five messages directed to separate core LRMs), and the actual amount of data sent as handshakes was further reduced by the use of very small handshake messages (see section 9.3.2). On the other hand, the data messages exchanged between particular applications and the gateway module were based on direct host to host communications, and as such were not received by any other modules. It should be noted that the ports used for broadcast and direct communication differed to allow appropriate separation of both streams of data.

The required datagram sockets were created with the use of the **socket** system call, their relevant options were set with the **setsockopt** routine, and finally the sockets were bound to appropriate network addresses and host ports (see Code example 9.2 below).

```
/* Creating a datagram socket */
*b_socket = socket(AF_INET, SOCK_DGRAM, 0);

/* Enabling broadcasting on the socket */
if (setsockopt(*b_socket, SOL_SOCKET, SO_BROADCAST, &on,
     sizeof(on)) == -1){
   perror("Set broadcast mode for socket failed");
   exit(1);
}

mbuff_size = MAX_MSG_LEN; /* Variable to hold the buffer size */

/* Setting send buffer size for the socket */
if (setsockopt(*b_socket, SOL_SOCKET, SO_SNDBUF, &mbuff_size,
     sizeof(mbuff_size)) == -1){
   perror("Set buffer size for send failed");
   exit(2);
}

/* Setting receive buffer size for the socket */
if (setsockopt(*b_socket, SOL_SOCKET, SO_RCVBUF, &mbuff_size,
     sizeof(mbuff_size)) == -1){
   perror("Set buffer size for receive failed");
   exit(3);
}

/* Setting address structure for the socket */
bzero((char*)&address, sizeof(address));
address.sin_family          = AF_INET;
address.sin_port            = htons(BROADCAST_SOCKET_PORT_NUM);
address.sin_addr.s_addr     = inet_addr(CORE_ADDRESS);

if (bind(*b_socket, (SOCKADDR *)&address, sizeof(address)) ==
   ERROR){
     perror("Broadcast socket - bind failed");
     exit(4);
}
/* The socket is now created and bound to its address and port */
```

**Code example 9.2. Broadcast socket creation - workstation code.**[90]

Note that the buffer size for send and receive commands is set in the code example above. It has been found that in the case of the SDCL implementation and the VxWorks operating system, the frequent use of sockets (messages being sent and received in short time intervals) led to leaks in the system memory. This is expected to have been related to some inefficiencies in the operating system, that was unable to quickly allocate and de-allocate bigger parts of memory (no such problem was observed in the UNIX-based implementation, see section 9.4.2 for more discussion). Reducing the size of the send and receive buffers for all sockets led to similarly stable system behaviour in both configurations.

[90] See [60] for detailed discussion on IRIX network programming.

Once the sockets are created and bound the process can receive the data from the bus with the **recvfrom** system call (see Code example 9.3), and send the data onto the bus with the **sendto** system call (see Code example 9.4).

```
int app_get_message(char *data, int input_socket, struct
                    sockaddr_in *source){
   int   addr_len,              /* Filled address structure length */
         data_len;              /* Received data length */

   /* First check data availability with the select call */
   struct timeval timeout;      /* Timer structure for select call */
   fd_set mask;                 /* Set of file descriptors to check */

   timeout.tv_sec = 0;          /* Effective waiting time is set to 0
   timeout.tv_usec = 0;         /* i.e. select returns immediately */

   FD_ZERO(&mask);              /* Reset file descriptors mask so that
                          /* no sockets/files/pipes are set */
   FD_SET(input_socket,&mask);/* And then set it to the required
                              socket */

   /* Perform select to check if there is a message available
      on the socket described by the mask variable the return value
   of ERROR indicates no messages ready */
   if (select(input_socket+1,&mask, NULL, NULL, &timeout)==-1)
      return ERROR;
   if (!FD_ISSET(input_socket, &mask))
      return ERROR;

   /* Message must be ready now */
   /* Prepare relevant data structures to read the message */
   bzero((char*)source, sizeof(*source));
   bzero(data,MAX_MSG_LEN);
   addr_len = sizeof(*source);

   /* Receive the message */
   if ((data_len = recvfrom(input_socket, data, MAX_MSG_LEN, 0,
      (SOCKADDR *)source, &addr_len)) == ERROR)
      perror("recvfrom failed in app_get_message");

   return data_len;
}
```

**Code example 9.3. Application side routine to receive data from the bus.**

Note that in the case of the reconfiguration process the above routine was slightly modified to introduce an idle wait for arriving messages (the reconfiguration process can wait for a period of time if no messages are available), and also some code was added to verify that the received message was in the required format and that it came from a valid source. This did not lead to any changes in the way the process interfaced with the data bus, but merely added further checks to parse and validate the message.

```
int app_send_message(char *data, int output_socket, const char
                *address, int port){

   /* The address the messages is to be send to */
   struct sockaddr_in destination_address;

   /* Prepare the address to send the message to */
   bzero((char*)&destination_address,sizeof(destination_address));
   destination_address.sin_family          = AF_INET;
   destination_address.sin_port            = htons(port);
   destination_address.sin_addr.s_addr     = inet_addr(address);

   if (sendto(output_socket, data, strlen(data), 0, (SOCKADDR*)
      &destination_address, sizeof(destination_address)) ==
   ERROR){
      perror("app_send_message failed);
      return ERROR;
   } else return OK;
}
```

**Code example 9.4. Application side routine to send data onto the bus.**

When the process that used particular sockets was to be terminated, it needed to perform the **close** system call on all its sockets to allow the system to reclaim the relevant resources (this is further discussed in sections 9.3.5 and 9.4.1). The port numbers used by particular processes are shown in the Table 9.5 below.

| Port number | Process | Input | Output |
|---|---|---|---|
| 3001 | Avionics application | ✖ | ✓ |
| 3002 | Avionics application | ✖ | ✓ |
| 3003 | Avionics application | ✓ | ✖ |
| 3004 | Avionics application | ✓ | ✖ |
| 3005 | Gateway | ✓ | ✖ |
| 3006 | Gateway | ✖ | ✓ |
| 3007 | Reconfiguration process | ✓ | ✖ |
| 3008 | Avionics application | ✖ | ✓ |

**Table 9.5. Port numbers used for inter-module communication.**

Note that sockets bound to port numbers 3007 and 3008 operated as broadcast sockets. Also, the port numbers listed above must be available prior to the execution of the relevant software in order to allow it to execute properly (in the case of a busy port number the processes have been implemented to terminate automatically).

### 9.3.5. Process management

The way termination and activation of processes during reconfiguration was implemented differed between the two configurations.

In the case of the SDCL some special functionality of the VxWorks system was used to implement the task management routines. The **taskDelete** and **taskSpawn** system calls were used to respectively terminate and activate the avionics applications [56], [57]. To investigate a possible gain in speed of application switching, some additional functionality of the VxWorks system was used to pre-spawn the tasks (the **taskInit** and **taskActivate** system calls). In this implementation the application software is not only pre-loaded by the module, but also all the application processes are spawned and then suspended, ready to be activated with a quick **taskActivate** system call. However, although this closely matches some of the functionality of APEX as defined in [10], it has proven not to shorten the time required for application switching, which has to be considered as a rather surprising result possibly related to some inefficiencies of the VxWorks system.

In order to simplify the task of maintaining the reconfiguration software for multiple platforms (VxWorks and various UNIX clones), the **taskSpawn** and **taskDelete** VxWorks system calls have been provided in the UNIX-based configuration as subroutines written with the use of the **fork**, exec and **kill** system calls [58], [59] (see Code example 9.5 and Code example 9.6). Note, that the function types had to be identical between the two implementations in order to avoid problems during compilation, and therefore some of the subroutine parameters are not used although they are present in the declaration.

```
STATUS taskDelete(int pID){
   char command[256];

   sprintf(command,"kill -9 %d",pID);
   if (system(command) == -1){
      perror("Internal fault. Unable to kill the child process!");
      return !OK;
   }

   return OK;
}
```

**Code example 9.5. UNIX-based implementation of the taskDelete VxWorks system call.**

Subsequent changes to the workstation implementation of the application and reconfiguration software rendered the call to **taskDelete** unnecessary, as the pseudo-avionics applications have been modified to terminate on receiving an appropriate signal from the reconfiguration process. The new implementation avoided some problems observed previously with the reclamation of socket ports as free system resources (these are further discussed later in this section and in section 9.4.1).

```
STATUS taskSpawn(char *n, int a, int b , int c,
     int fn(int,int,int,int,int,int,int,int,int,int),
     int state_ptr, int h, int i, int j, int l, int p, int q,
     int r, int w, int x){

   pid_t processID;          /* ID of the newly spawned process */

   processID = fork();
   if (processID == 0){      /* 0 indicates the child process */
      char path[256];
      sprintf(path,"../unix/%s/apps",n);
      if (execlp(path, path, (char *)state_ptr, NULL) == -1){
         printf("Unable to start function: %s. Exiting...\n",path);
         exit(99);
      }
   }

   return processID;
}
```

**Code example 9.6. UNIX-based implementation of taskSpawn VxWorks system call.**

Commentary:
Note that in the above example the first of the function parameters indicates the application to be spawned. For example parameter n set to "app0" indicates that the executable called "apps" from the sub-directory "app0" should be activated (executables for particular functions were stored in separate directories). The state_ptr parameter indicates the place in the memory where the state of the application to be spawned is stored - the system will only allow the application to read the state and not to modify it, an attempt to do so would result in a segmentation fault and task termination. Finally, all other parameters present in the function declaration were added merely for the purpose of maintaining compatibility with the VxWorks implementation.

Some questions relating to memory partitioning should be asked when the passing of state information to the new application is being considered. In this implementation a pointer to a memory location containing the application state is passed to the new process, so the spawned application can initialise properly. This implies that some parts of the core LRM memory will be shared by more than one task, although only one task will have permission to write to it. In view of the ARINC 653 APEX specification the use of shared memory would not be allowed, and thus the reconfiguration process might need to send to the application a specific message to set its state. This would be relatively easy to achieve in the

implementation discussed in this chapter, as technically[91] there is nothing precluding the application from receiving the state messages and modifying its own execution accordingly. Only minor modifications would be necessary both to the reconfiguration and the application software to accomplish this task.

As all the applications used the same ports for their communication, it was observed that primarily in the case of the VxWorks system[92] the sockets used by the application being terminated needed to be closed before the actual deletion of the task, to allow the new application to reuse the ports. Therefore, in both implementations an additional signal was sent by the reconfiguration process to indicate to the application that it should close its sockets as it is being terminated. The application software needed thus to be modified to be compatible with the reconfiguration process[93], and in the case of the UNIX configuration, on receiving the signal the application process was not only required to close its sockets, but also as soon as this has been completed to terminate itself (hence no need for the **taskDelete** call).

Despite the fact that the applications software was retrieved from the disk each time a function was spawned in the UNIX configuration, it was observed that it required less time to switch the pseudo-avionics applications (note that in the SDCL all the software was pre-loaded, and thus present in the module memory when the task was being spawned). This could be attributed to the more powerful CPUs of the workstations used and a more efficient and faster operating system.

### 9.3.6. Timing

As mentioned before, in the case of the SDCL implementation the internal system clock was used to time stamp received messages and to maintain the rate at which messages were sent. The clock operated with 60 Hz frequency, giving 16.67 ms to a clock tick. As the tick count is set to zero at the start of the reconfiguration process (the **tickSet** VxWorks call is performed), and it is a 32 bit unsigned integer, the

---

[91] Some additional integrity issues may arise at this point, as the possibility of modifying the execution of an application would have to be restricted only to the period of time shortly following its activation.

[92] This behaviour could be sometimes observed on a UNIX-based workstation, should the system resources or its CPU be occupied by other users or services.

[93] This involved just a few lines of code.

scheme based on such a counter would be able to operate without being reset for some 828 days. The current tick count was obtained with the **tickGet** system call.

In the case of the UNIX configuration the **tickGet** and **tickSet** functions have been emulated with the use of the **gettimeofday** system call (see Code example 9.7). The **gettimeofday** call returns the number of seconds and the number of microseconds elapsed since midnight (00:00) Co-ordinated Universal Time (UTC), January 1, 1970. Based on this information, an incremental system clock has been implemented with a tick length of 15 ms[94] to emulate the functionality of the SDCL-based system and to enable the use of the software on different platforms. As the tick counter used on the UNIX systems is a signed 64 bit integer, the maximum length of the scheme continuous operation before the counter exceeds its upper limit is somewhat longer than four billion years.

```
#define TICK_LEN 15000   /* Tick length in microseconds = 15ms */

/* Zero set to the gettimeofday seconds counter at the beginning of
operation to keep tick counts similar to VxWorks - starting from
approximately zero and then increasing */
unsigned long zero_cnt=0;

/* Structures required to perform time reading by gettimeofday */
struct timeval tick_tv;
struct timezone tick_tz = { 0, DST_GB };

unsigned long tickGet(){
  gettimeofday(&tick_tv, &tick_tz);
  return ((tick_tv.tv_sec - zero_cnt) *
            1000000 + tick_tv.tv_usec) / TICK_LEN;
}

unsigned long tickSet(unsigned long new_cnt){
  gettimeofday(&tick_tv,&tick_tz);
  zero_cnt = tick_tv.tv_sec - new_cnt;
  return new_cnt;
}
```

**Code example 9.7. Workstation code for emulation of the VxWorks tick counter.**

Note that for simplicity **tickSet** uses only seconds and not microseconds to reset the timer. This, however, does not introduce any problems with the implementation of the scheme as it does not depend in any way on the tick counter being reset exactly to zero or even being reset at all.

[94] Giving a clock frequency of 66.667 Hz.

## 9.4. Discussion

This section discusses problems and issues identified during the implementation phase. Although as mentioned before, neither of the systems used was totally representative for implementation of RIMA (particularly with respect to the use of an asynchronous data bus as a backplane), certain aspects should remain valid for a wide range of possible hardware and software platforms.

### 9.4.1. Process management

In the case of the VxWorks system some rather undesirable system behaviour was observed when task termination and task spawning were being investigated. Despite the fact that the Tornado system was designed as a real-time system, the time required for reconfiguration (termination of the current function and activation of a new application) was observed to vary from two to nine clock ticks. It is difficult to explain what caused such a significant variance, as the system load related to running the processes and interfacing the backplane bus was comparable in all cases. Furthermore, it was rather surprising that reconfiguration based on pre-spawned processes (see section 9.3.5 for details) was not faster than that based only on pre-loaded software, which led to further questions about system efficiency. This, combined with the lack of memory partitioning between tasks in the VxWorks Tornado system indicates that the OS should not be considered for commercial implementation of RIMA.

It has been observed in the UNIX-based configuration that the system was much faster with respect to terminating and spawning new processes than it was in the SDCL case. This was even more unexpected as the applications software needed to be downloaded to the module memory from a disk drive before the avionics function process could be started. Despite the necessity for software download it has been found that the whole process of reconfiguring an application was normally taking two clock ticks (30 ms), and was never longer than three clock ticks.

As all applications used the same port numbers for their communications some problems have occurred related to such a rapid reconfiguration. Namely, on several occasions the system was unable to reclaim the freed ports, although the application had closed its sockets, and it did not allow the new application to open the communication sockets with the same port numbers. To eliminate the problem, the

reconfiguration process had to be put into an idle delay to allow the system to duly notice that the ports are again available for allocation. As this problems did not occur systematically, it is expected that they were related to the fact that the machine CPU might have been used by other users or kernel services in the system (as mentioned before the software was run on an open system available to other users). In an actual avionics system where all the CPU power can be accounted for, this should not be the case, especially if some spare processing capacity exists within the system. Similarly, the favourable performance of the UNIX-based workstations could possibly be explained by the high capacity of the machines CPU and a very fast and stable multitasking operating system.

### 9.4.2. Data bus problems

Several problems were encountered whilst interfacing the backplane data bus and conducting inter-module communications in either of the configurations. They relate to the asynchronous access protocol the data buses were running, and to the way the OS was handling the messages.

It has been observed that in both configurations the order and timing of the messages sent by the processing modules could not be guaranteed. The data bus asynchronous mode of operation indicated that messages will sometimes have to be buffered by the OS while it was waiting for the bus access. The buffering and other related delays clearly depended on the frequency the messages were being sent

In order to meet the real-time requirements of critical functions the relevant messages needed to be sent with every system clock tick, i.e. every 15 ms or 16.67 ms. The length of the time the messages were delayed or buffered depended also on the spare processing capacity of the operating system and its ability to efficiently interface the bus. Again, these have proven to be shorter for the UNIX-based configuration.

As the messages were delayed before they could be transmitted by particular processing modules, it has been observed that the sequence in which they were appearing on the data bus could not be pre-determined. Moreover, in the case of the SDCL configuration where an FDDI ring was used as a backplane, the order in which the messages were received also differed from one module to another. This was related to the fact that all messages were passed host to host on the FDDI ring before they reached their destination, and thus different delays will have been induced on messages arriving at different

hosts.[95] Such unfavourable behaviour of the ring based data bus indicates that similar solutions are unlikely to be suitable for implementation of dynamically reconfigurable systems, as although it is possible to design reliable broadcast algorithms for such systems [62], they would be exposed to experiencing even longer and thus unacceptable delays.

Some additional problems have been observed when real-time reconfiguration of critical functions has been considered. As the messages were buffered and delayed in a non-deterministic fashion, it was difficult to achieve reliable behaviour of the reconfiguration scheme operating with short failure delays. To meet the previously assumed 400 ms reconfiguration requirement for critical functions with four backup modules, the scheme would have to operate with at most six consecutive messages used for failure detection and sent every clock tick.

However, although the analysis from Chapter 8 identifies the probability of invalid reconfiguration of such a scheme as much lower than extremely improbable, in the case of unreliable and non-deterministic communication media with delays induced on particular messages, it has been practically observed that spurious reconfiguration may occur. This refers particularly to the rather inefficient SDCL configuration which was unable to operate properly based on less than some 10 messages per failure detection. The more capable UNIX-based configuration was able to meet the requirements only if both the OS and the bus were not in use by other users or services. Note that the author of [40] claims that "timeouts and other time-based protocol techniques are possible only when a system is synchronous", which again underlines the importance of a deterministic backplane bus, possibly based on the ARINC 659 standard or its modification. It would be feasible to implement timeout based systems employing asynchronous communication media provided that the upper bound on the communication delays can be identified [63]. In principle, this implies that the observations of the reconfiguration scheme are consistent with the theory and they indicate that the upper bound on the message delays in the SDCL-based configuration was higher than in the UNIX-based configuration.

---

[95] Notably the problem did not occur in the UNIX-based configuration utilising an Ethernet LAN as the communication medium, that is mentioned in [61] as a broadcast capable network.

Additional problems with memory management were encountered on the SDCL configuration when the VxWorks system was forced to quickly allocate and de-allocate buffers for the messages being received and sent. The problem exhibited itself as a slow but consistent loss of some of the system memory, which in the long term led to the system being halted. This was rectified by modifying the buffer size for the datagram sockets making it easier for the system to allocate and to later reclaim the memory.

The default buffer size for the sockets used in the implementation was pre-defined in the system as 2048 bytes, whereas the maximum message length used in reconfiguration was set to 512 bytes (see Code example 9.2). After reducing the size of the receive and send buffers, the VxWorks Tornado system did not experience any further loss of system memory, and was able to operate properly and not halt even after long periods of time. Note that this problem did not occur in the UNIX-based implementation, which may indicate again greater efficiency and stability of the SGI Irix and the Linux operating systems.

Finally, as the Internet Protocol (IP) underlies communications in both cases, some additional software version management efforts were required to provide avionics applications for different modules. Namely, each of the core LRMs connected to the data bus was allocated a different IP address, making it distinct from all other modules. In order to be able to open non-broadcast communication sockets the application software needed to bind them to an IP address - the address of the machine the software will be running on. This clearly indicates that a different version of each application is required for each core LRM, implying a great multiplicity of software. However, this problem is relatively simple to rectify in one of the following ways:

- setting all sockets as broadcasting sockets makes the software independent from the core IP address (this was the case for the reconfiguration process which utilises broadcasting sockets)
- the IP address for each module is defined in a header file, and thus in order to obtain a version for a different machine only a simple change of a single constant and subsequent recompilation is required (this was the case for the avionics applications and the gateway software).

In conclusion, it has to be emphasised again that the asynchronous data buses used in both systems were far from ideal for implementation of RIMA, and alternatives should be sought (e.g. real-time

communication over LAN [63]). A valuable insight into applicability of a modified ARINC 659 standard based data bus to reconfigurable avionics can be gained from [42].

### 9.4.3. Time constraints and real-time operation

It is vital that the behaviour of the reconfiguration process is perceived in terms of real-time operation. In the case of the systems used for implementation of RIMA certain problems with deterministic scheduling and time sharing have been observed, which should be taken into account when designing a commercial application of RIMA. Some of the problems relate more to the features of the operating systems used for implementation (particularly the VxWorks Tornado system); others are inherent to the design of the scheme (e.g. CPU requirements for message handling with a large number of packets sent in short time intervals).

Although the VxWorks Tornado system asserts to be a real-time OS, it has been observed that it is highly inconsistent in terms of the time required to perform particular tasks. Specifically, the time required to terminate an application and to spawn a new one was seen to vary between two and nine clock ticks. Such behaviour should be considered highly unsatisfactory (450% difference between the minimum and the maximum time required), as it renders all considerations about the possible system performance speculative and unreliable. Moreover, with reconfiguration taking up to nine clock ticks (some 150 ms), it will be impossible for the system to meet the 400 ms requirement for reconfiguration of critical applications with multiple backups.

Contrary to the SDCL system, all workstations in the alternative configuration exhibited very consistent behaviour, requiring at most three clock ticks for reconfiguration. The variance of the reconfiguration time between two and three ticks can be easily explained with the fact that the process may start at any point during the tick, and therefore the time measured in ticks can vary by one unit (see Figure 9.1 for illustration).

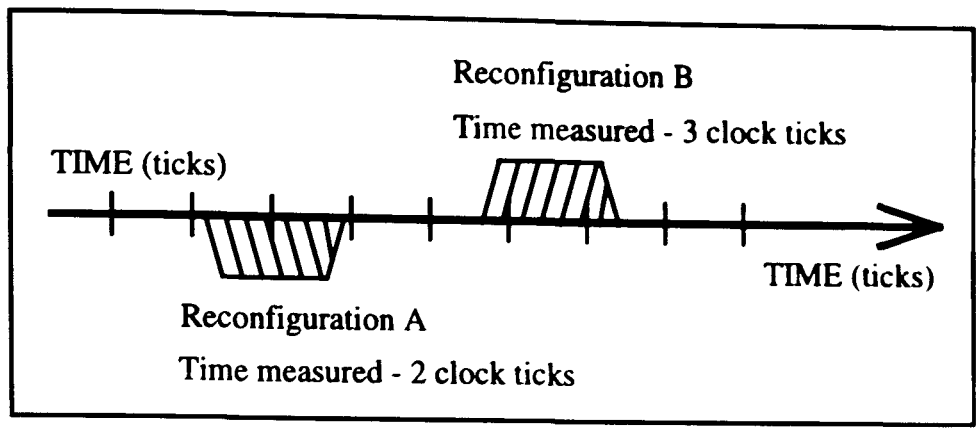



**Figure 9.1. Expected variance in timing of application reconfiguration.**

Another important issue relates to the failure detection phase. In order to be able to operate reliably, the failure detection mechanism must be based on multiple messages. It has been calculated, in Chapter 8, that in implementations based on sufficiently reliable hardware as few as two messages can be used to obtain reliable failure detection. However, as already discussed in sections 9.3.4 and 9.4.2, due to the asynchronous media and inconsistent message handling by the OS, spurious reconfiguration could occur even with many more messages. This makes it again difficult to analyse at what point the behaviour of the scheme becomes stable and reliable, particularly as both the OS and the CPU itself might be occupied by independent tasks embedded into the OS or supplied by other users.

In the case of the UNIX-based configuration it has been found that with an idle system (no external users and minimum of independent services), the failure detection mechanism can operate properly on as few as six messages giving the failure detection delay of 90 ms. With the reconfiguration delay of approximately 30 ms such a system could operate within the 400 ms reconfiguration requirement for the critical applications. On the other hand, in the case of the SDCL configuration, the least number of messages required for reliable failure detection was extremely difficult to establish, possibly due to the inefficient OS kernel giving very inconsistent behaviour. It can be estimated, that as the failure delay needs to be longer than the reconfiguration delay, the scheme would not be able to operate properly based on fewer than ten messages per failure detection (in the case of critical applications the messages were sent every clock tick). In practice, this had to be set to a much higher value of fifteen messages to allow for system inefficiency with handling frequent data bus communications.

Certain real-time issues arise with respect to the way the scheme handles the communications, and in particular to the operating system ability to cope with a great number of packets being sent to and

received from the backplane bus (see Table 9.4 for details). It was observed that the messages were buffered by the system before they were being made available for the reconfiguration process, the applications and the gateway. This again questions all the assumptions on the timing of the failure detection mechanism, but can be relatively easily solved by the use of an alternative data bus.

For example, a modified ARINC 659 standard data bus could be employed for reliable and quick failure detection as briefly discussed in [42]. In this case, the amount of data on the backplane bus would also be greatly reduced, as the reconfiguration software could easily monitor data presence in application transmitting windows, and thus no specific handshake messages would be required to identify whether or not the function is being performed. This would clearly reduce the CPU requirements for the applications, and would eliminate problems related to operating system inability of coping with a high number of packets arriving in short time intervals.

## 9.5. Conclusions

The work conducted on both systems has allowed identification of a number of issues which need to be resolved when an actual commercial system is to be implemented. The two major areas where some undesirable behaviour has been observed relate to the data bus used as the backplane, and to the OS capacity for real-time operation and real-time handling of frequent communication messages.

It has been observed that asynchronous data buses are extremely unlikely to be a viable option for implementation of backplane communications within a cabinet. The non-deterministic character of the data bus access protocol makes the failure detection phase less reliable, there being no guarantee how long the messages may get delayed on either end of the connection, and thus it must operate with longer delays. This clearly has an impact on the real-time performance of the scheme, and thus renders non-deterministic asynchronous data buses as a rather unattractive choice for implementation of the cabinet backplane.[96]

Further more, in the case of an FDDI ring also the order in which the messages are received by particular processing modules cannot be guaranteed, and it may differ between the cores. This introduces additional non-deterministic factors making the analysis of the scheme operation very difficult.

Other problems have been encountered with respect to the real-time operation of the operating systems, particularly in the case of the VxWorks Tornado system. It has been observed that the system was not powerful enough to properly handle frequent data bus communications within reasonable time constraints. It is expected that it was the system rather than the data bus access protocol that was contributing the greatest delay on messages as they were buffered. Additional problems have been encountered with the timing of particular services (e.g. termination or activation of applications), which were highly inconsistent and thus introduced additional non-determinism into the scheme. From the three systems used (VxWorks Tornado, Redhat Linux and SGI Irix), it was the latter two which performed reliably, possibly because of the greater power of the CPUs running the OS.

Despite the problems discussed above, the implementation phase allowed the investigation of the behaviour of the reconfiguration scheme in real-time operation as a distributed system (separate units were running different applications). Although the avionics software only simulated the performance of the actual avionics functions, its behaviour with respect to the reconfiguration scheme can be considered as indistinguishable from that of the real applications (the relevant messages were produced in rigidly maintained time intervals). Furthermore, in the case of the UNIX-based configuration, even the problems with the non-deterministic data bus could to some extent be solved with longer failure delays, and the scheme was able to operate within reasonable time constraints. It was impossible to obtain similar behaviour from the SDCL configuration.

In order to fully test the practical feasibility of autonomous dynamic reconfiguration within an avionics system, an implementation based on deterministic backplane bus and strict real-time operating system would be required. However, some encouraging results of the demonstration were obtained even with only a partially representative system.

---

[96] In [63] the author discusses the use of an asynchronous local area network as the basis for an implementation of reliable real-time communications, however, detailed information about expected data traffic, expected processing

power requirements etc. is necessary to implement the algorithm.

# Chapter 10. Certification Issues

## 10.1. Introduction

In previous chapters various aspects of reconfigurable avionics systems have been presented and discussed. This involved the definition of a suitable hardware architecture, the proposal of an autonomous reconfiguration scheme, its formal specification and practical demonstrations on two differently configured systems.

The experience gained from practical demonstration and various requirements identified prior to that phase were intended to provide a good insight into safety related risks following the introduction of reconfiguration as means for providing system fault tolerance. In this chapter, issues related to future certification of reconfigurable avionics are gathered and presented with appropriate references to chapters and sections where they were discussed in detail.

## 10.2. Hardware related issues

This section focuses on various certification issues referring to safety hazards related directly to the hardware specification of the actual system.

### 10.2.1. The backplane data bus

It has been identified during implementation of a reconfigurable system that the issue of the backplane bus mode of operation, its throughput and access protocol are of great importance, particularly when the reliability and timing of the failure detection mechanism are in question.

One of the first problems identified in this area relates to the asynchronous mode of operation of the data buses used by both SDCL and the University based system. It has been observed that it is extremely difficult to guarantee that the failure detection mechanism will meet its timing requirements, should it be based on the monitoring of an asynchronous data bus. This has been discussed in detail in Chapter 9, and

suggests that backplane bus synchronicity is highly desirable, as it allows the system integrator to accurately predict its behaviour.

It should be noted at this point that the backplane bus standard (ARINC 659) - proposed in IMA architecture examples as seen in the ARINC 651 report - does not allow for free dynamic changes of the data bus access schedule, and as such appears to be unsuitable for reconfigurable systems. However, should the standard be modified to allow identity changes (for example similar to these discussed in [42]), its derivatives could be considered for implementation in RIMA.

As discussed in Chapter 5, in order to minimise the length of reconfiguration delays the software for avionics applications should be stored internally by particular core LRMs. This renders the issue of the backplane bus throughput somewhat secondary to its mode of operation. However, should the reconfiguration scheme be based on handshake messages as opposed to monitoring for the actual data or possibly the change of the freshness flag in the transmitting window assigned to particular functions, the problem of a great number of small messages being exchanged on the data bus could arise.

It was observed during the implementation phase, that the processing units will require great capacity for interacting with the data bus should the handshake messages be used. Moreover, in such an implementation the data bus itself must allow operation with possibly several hundreds of small messages exchanged every second (see Table 9.4 for the number of messages used in the implementation of a five core cabinet). Thus, it appears that methods alternative to the monitoring of handshake messages should be recommended for implementation of the failure detection mechanism in order to simplify the data bus requirements, conformance with which might be difficult to show otherwise.

### 10.2.2. The CPU

As discussed in Chapter 9, the issue of the processing capacity of the core LRMs might prove vital when conformance with the real-time requirements of the system is being considered. Although it has been shown that the actual processing power requirements related to the avionics application, the reconfiguration process or the gateway software should not be difficult to satisfy, the problem of interfacing the data bus might complicate the matter.

In the case of the SDCL-based system the problem manifested itself with some problems related to invalid failure detection, as the module was unable to retrieve a great number of messages from the data bus sufficiently quickly. In the case of the UNIX-based configuration this problem did not occur, probably due to more a powerful CPU and a greater efficiency with which the operating system was dealing with the backplane bus traffic.

It should be noted that the processing power requirements should be relatively easy to satisfy should the failure detection mechanism be based on alternative methods such as the freshness flag monitoring in the module transmitting window. It is expected that such an approach would not involve excessive computation,[97] although a detailed analysis would be required when the final system architecture is properly defined.

### 10.2.3. Memory

The possibility of spontaneous changes of memory location occurring at any time during the system operation raises some questions about the possible corruption of the reconfiguration data, reconfiguration or application software and finally the application state. Although this is difficult to predict or control, the reliability of the memory chips may impose additional requirements on the reconfiguration scheme and the consistency maintenance procedures.

[97] This assumes that the system can efficiently monitor the freshness flag or the presence of new data in all transmitting windows within each frame.

| Subject | Certification issues | Section |
|---|---|---|
| Mode of operation of the backplane bus | synchronous mode of operation of the backplane bus is a key issue in the analysis of the failure detection mechanism | 9.4.2 |
| Backplane bus access protocol | the backplane bus should guarantee timings of messages sent by particular core LRMs, so that the messages are not buffered for varying periods of time before appearing on the data bus | 9.4.2 |
| Backplane bus throughput | guarantee that the backplane bus will be able to operate with a great number of handshake messages being exchanged in short time intervals, should the reconfiguration scheme require the failure detection mechanism to be based on handshake messages | 9.3.4<br>9.4.2 |
| CPU | guarantee that the CPU will provide sufficient power to deterministically handle data bus and process management requests within the time constraints | 9.4.1<br>9.4.3 |
| Memory | guarantee that spontaneous changes of memory locations will not affect stored data or software beyond the permissible limits<br><br>note that this should be considered in the context of the data maintenance procedures | 5.2.1.6<br>10.2.3 |

**Table 10.1. Hardware related certification issues.**

## 10.3. Operating system and application executive related issues

It became clear during the demonstration phase that the lack of certain OS/APEX services can render the reconfigurable approach to system design unfeasible. Moreover, should the services be available but their implementation inefficient, the reconfiguration process may not be able to operate correctly or within the time constraints. In this section several services required to implement the scheme and their safety aspects will be discussed in the areas of process management and data bus access. Note that the discussion will focus on these aspects of the services which relate directly to reconfiguration.[98]

### 10.3.1. Process management

The issue of integrity of the process management routines relates to

- predictable real-time scheduling
- task activation and termination
- memory management.

[98] A discussion of other safety related issues of OS/APEX services can be found in [10].

It has been found during the implementation of RIMA in the SDCL configuration, that the operating system was unable to guarantee the time required for performance of particular functions. This clearly makes it extremely difficult to guarantee that the reconfiguration algorithm will execute within its real-time constraints. Therefore, deterministic and predictable scheduling of tasks by the operating system or the application executive should considered as a certification issue.

Furthermore, as the reconfiguration process is allowed to terminate and to activate the applications, the integrity of implementation of appropriate routines must be guaranteed. This issue also arises when implementation of the reconfiguration software itself is in question, and it is discussed later in this chapter.

The ARINC 653 APEX standard does not make provisions for the use of shared memory (possibly based on the expected safety risk related to the situation where multiple processes access the same memory locations), thus the issue of restarting an application from a given state must be resolved. This could involve sending an appropriate message to a running application and allowing it to modify its execution. Again, the integrity of the relevant mechanisms should be investigated, as any unwanted changes to the state information may cause the application to execute improperly. This could relate to the way the state is stored in the module memory or the way it is handled by the communication routines. The issue of state also arises when the reconfiguration process and the avionics applications are discussed.

### 10.3.2. Data bus access

Based on the comparison between various operating systems and their performance with respect to accessing the data bus, it has been concluded that the efficiency and the throughput capacity of appropriate services must be questioned when implementing a reconfigurable system. Even with solutions based on monitoring of the freshness flag as opposed to monitoring of the data it can be expected that any inefficiencies in the communication routines may lead to invalid operation of the failure detection mechanism, and as a result to the failure of the reconfiguration scheme itself.

With respect to the failure detection mechanism, the communication routines should guarantee the timing of particular messages. It was observed during the implementation phase, that messages sent by particular applications where buffered for a varying period of time before they have appeared on the backplane bus, thus rendering the time constraints conformance analysis of the failure detection mechanism extremely difficult. Therefore, the issue of deterministic data bus access should be again considered a certification issue when the implementation of the communication services is being considered.

### 10.3.3. Clocking device

In order for the failure detection mechanism to be able to operate correctly, the system must provide an incremental clock used for time stamping the messages. Such a clock device must guarantee that the value of the clock will not exceed its upper limit within a reasonable period of time, normally understood as the longest system run without reset.

| Subject | Certification issues | Section |
|---|---|---|
| Scheduling of tasks | guarantee of deterministic and consistent scheduling of particular tasks | 9.4.3 |
| Execution of tasks | guarantee of the length of the period of time required to execute particular tasks<br>guarantee of consistency of task execution in terms of the time required to complete the task | 9.4.1<br>9.4.3 |
| Activation and termination of applications | guarantee of integrity of appropriate routines and system calls | 10.3.1 |
| Integrity of state communication | guarantee that state can and will be communicated properly to the desired application | 10.3.1 |
| Data bus communications | guarantee of predictability and consistency of execution of the data bus access routines and services, with particular emphasis on the capacity for backplane bus monitoring | 9.3.4<br>9.4.3<br>10.3.2 |
| Incremental clock | guarantee that the incremental clock device will not reach its upper limit during normal system operation | 9.3.6<br>9.4.3<br>10.3.4 |

**Table 10.2. OS/APEX related certification issues.**

## 10.4. The application software

The interaction between the reconfiguration process and the avionics applications is limited to their activation and termination. In essence, the application software must allow the reconfiguration process to execute the function from any given state[99], and to allow it to terminate the application and to reclaim its resources. Thus the avionics software must guarantee that it is able to re-execute from a desired checkpoint, and that it will free its resources on termination.

## 10.5. Reconfiguration scheme

The design principles for reconfiguration schemes have been discussed in detail in Chapter 5, and an example of one has been defined and formally specified in chapters 7 and 8. Based on the previous discussion, various safety related aspects of reconfiguration software in RIMA systems are gathered in this section in order to identify associated certification issues.

### 10.5.1. Dispatch with known failure

When considering the certification of RIMA systems within the current procedures, one should notice that in order to allow the system to fully benefit from its capacity for dynamic reconfiguration, the requirement forbidding "dispatch with known failure" should be modified or abandoned. The study in Chapter 4 shows that RIMA systems can provide high availability, however, they must be allowed to be dispatched with some of the modules failed. Note that failures of some core modules in the cabinet do not affect the aircraft safety – the philosophy of RIMA is based on shared redundancy and expendable redundant modules – and thus RIMA systems should not be made to adhere to the current regulations.

### 10.5.2. Other certification issues

The following Table 10.3 provides a compilation of such issues and provides reference to appropriate sections, where they are discussed in detail.

---

[99] The issue of application state and its identification is being widely researched within the computer science community, see for example [23], and will not be discussed at this time.

| Subject | Certification issues | Section |
|---|---|---|
| Dispatch with known failure | guarantee that the system is safe to be dispatched with some of the core LRMs failed<br><br>identification of the actual system tolerance in terms of the highest number of failed modules that the system can be dispatched with | 4.2<br>10.5.1 |
| Reconfiguration delays | assessment of time delays related to event detection and reconfiguration for the "worst" case scenario of combinations of failure and recovery events | 5.2.2.1 |
| Reconfiguration chains | assessment of the length of reconfiguration chains for the "worst" combinations of failure and recovery events<br><br>proof that none recovery event of a core LRM will be followed by reconfiguration of other module(s) | 5.2.2.2<br>8.2<br>8.3 |
| Reconfiguration algorithm determinism | proof of algorithm property of at least normal determinism<br><br>formal verification of the reconfiguration scheme determinism | 5.2.3<br>8.3 |
| Reconfiguration algorithm integrity | guarantee that when required reconfiguration will take place, and will relate only to modules that satisfy activation conditions<br><br>note that assurance of integrity of appropriate system services should also be required in order to be able to guarantee that the reconfiguration algorithm will maintain reliable process management | 5.2.3<br>8.2 |
| Independence of core LRMs | guarantee of independent operation of processing modules, in order to eliminate problems related to fault propagation | 5.2.1.3 |
| Equality of core LRMs | guarantee that neither of the core LRMs can force its decisions (actions) on other processing modules (similarly to the point above) | 5.2.1.3 |
| Reconfiguration phase-synchronisation | identification of core LRMs behaviour in the event of missed or misunderstood messages (events)<br><br>analysis of situations where a core LRM or core LRMs wrongly interpret the state of the cabinet | 5.2.1.2<br>8.5 |
| Reconfiguration activation conditions | assurance that activation conditions must be satisfied prior to reconfiguration<br><br>assurance of integrity of the activation conditions evaluation routines | 5.2.1.4<br>5.2.1.5<br>5.2.1.6<br>8.2 |
| Invalid activation of reconfiguration | identification of sources of possible invalid activation of the reconfiguration process<br><br>specification of system behaviour on invalid activation of the reconfiguration process | 5.2.1.5<br>8.5<br>8.7 |
| Reconfiguration data<br><br>(strategy and | proof of integrity of the reconfiguration data<br><br>maintenance of reconfiguration data | 5.2.1.6<br>5.2.7.1<br>6.3.2<br>8.4 |

| | | |
|---|---|---|
| auxiliary data) | consistency<br>specification of reconfiguration strategy data update methodology (if any) | |
| Algorithm corruption | identification of sources of algorithm corruption including corruption of reconfiguration data<br>analysis of behaviour of the reconfiguration and recovery algorithms while operating with corrupted data | 5.2.7.1<br>5.2.7.2<br>5.2.7.3<br>8.7 |
| Failure detection | definition of a failure and its manifestation<br>specification of the failure detection mechanism<br>identification of the length of the delays related to backplane bus monitoring<br>analysis of failure detection reliability in terms of undetected failures or detection of events wrongly classified as failures (note that the backplane bus specification needs to be known at that stage)<br>identification of algorithm behaviour in the case of missed or misunderstood data or messages<br>identification of possible sources of invalid failure detection and related failure conditions<br>analysis of conditions classified as simultaneous or nearly simultaneous events | 5.2.2.1<br>5.2.3<br>5.2.4.1<br>5.2.4.2<br>5.2.4.3<br>5.2.4.4<br>8.3<br>8.5<br>8.7<br>9.4<br>10.2.1<br>10.3.2 |
| Processing module recovery<br>(if supported) | specification of recoverable failures<br>specification of the recovery mechanism<br>proof of integrity of the recovery mechanism | 5.2.5.1<br>5.2.5.2<br>8.2.4<br>8.2.5 |
| Recovery detection<br>(if recovery supported) | justification of the lack of recovery detection<br>or<br>if recovery detection is required, same as for the "Failure detection" entry | 5.2.5.1<br>5.2.5.2<br>5.2.5.3<br>5.2.5.4 |
| Failure and recovery related actions | specification of actions that will be taken by the reconfiguration and recovery algorithms in the case of a failure or a recovery event | 5.2.4.4<br>5.2.5.4 |
| Fault indication | specification of fault communication mechanism between the cabinet and the crew | 5.2.8 |
| Function state updating | specification of the avionics function state updating approach<br>specification of the coherency maintenance method for multiple states of a single avionics function<br>assurance of integrity of state maintenance and state transfer | 5.2.9<br>10.3.1<br>10.4 |
| Software version control | specification of appropriate procedures for maintaining multiple versions of the reconfiguration software | 9.4.2 |
| Time constraints conformance | guarantee that the reconfiguration scheme will meet the time constraints imposed by the safety requirements with respect to the | 8.5<br>8.6 |

| | | |
|---|---|---|
| | timing of failure detection and reconfiguration | 9.4 |
| Reconfiguration scheme properties of safety | proof of the reconfiguration scheme properties critical to the system safety | 9.2 |

**Table 10.3. Compilation of the reconfiguration scheme related issues**

# Chapter 11. Conclusions and Recommendations for Future Work

## 11.1. Introduction

The main project objectives were to establish possible benefits related to reconfiguration in avionics systems, and to assess the feasibility of implementation of an autonomous dynamic reconfigurable system. The sponsors of the research- the UK Civil Aviation Authority were further interested in identifying RIMA related certification issues, that had been discussed in the previous chapter.

This chapter concludes findings related both to the benefits and the feasibility study, as well as providing some recommendations for future work in this field.

## 11.2. Benefits of RIMA

It has been shown in Chapter 4 that reconfigurable avionics systems exhibit a great potential for operation with significantly reduced processing module redundancy, when compared with non-reconfigurable systems. The results presented by Figure 4.3 and Figure 4.4 indicate that RIMA can operate within strict safety and availability requirements with the processing modules redundancy tenfold lower than in the case of an equivalent non-reconfigurable system.

Moreover, further analysis showed that RIMA systems are able to achieve the "C-check" of 3,000 hours with the 99% probability that no unscheduled maintenance will be required. This additional study was conducted after talks with British Airways, who suggested that it would be preferred from the aircraft operator point of view, if the 400 hours availability objective was extended to 3000 hours. Although the redundancy of processing modules had to be increased due to more demanding dispatch availability requirements, RIMA systems again compared favourably with the non-reconfigurable approaches (see Figure 11.1 and Figure 11.2 for the actual results calculated for processing units of 30,000 hours MTBF each).

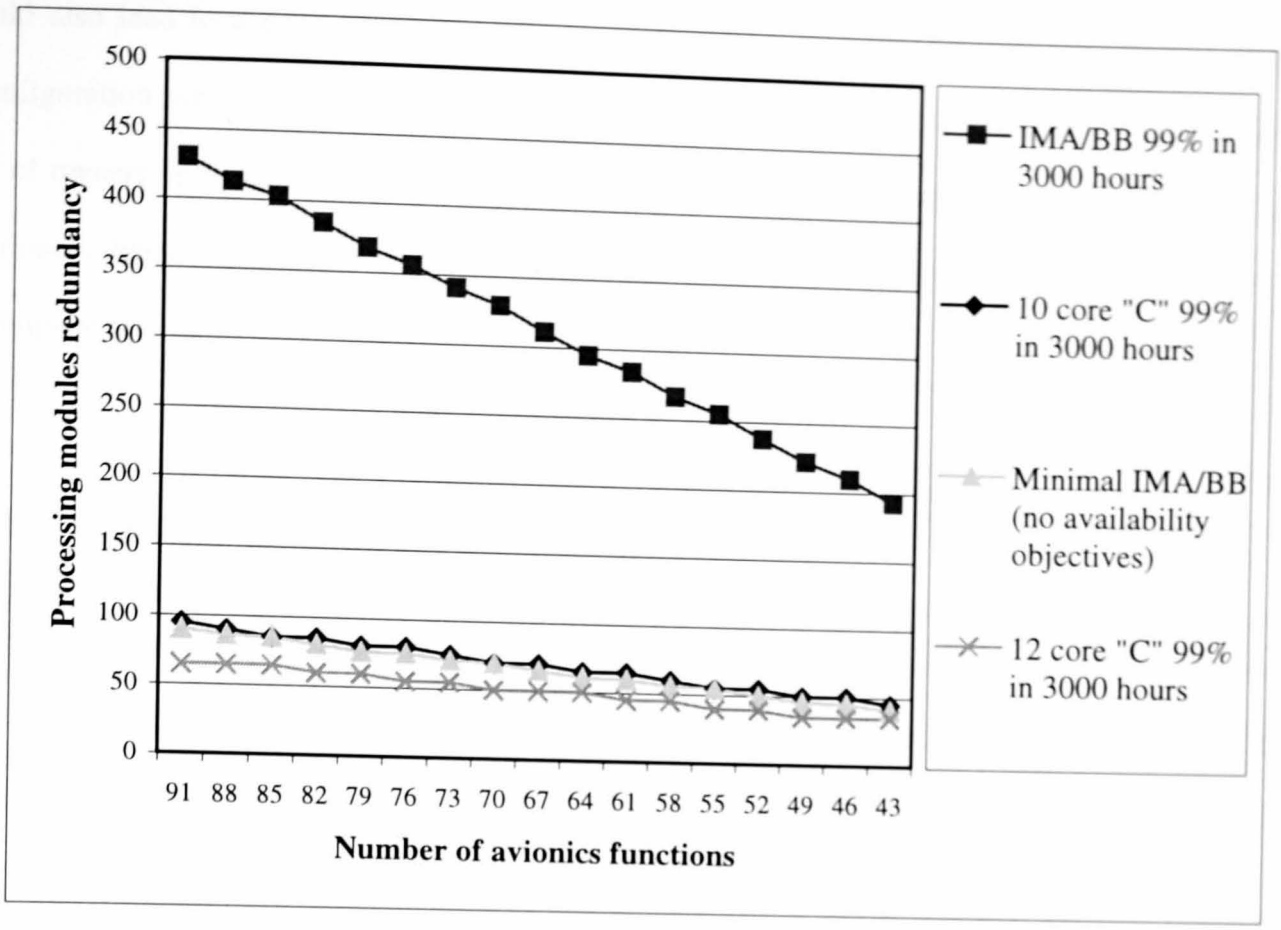



**Figure 11.1. Core LRM redundancy figures for "C-check" systems.**

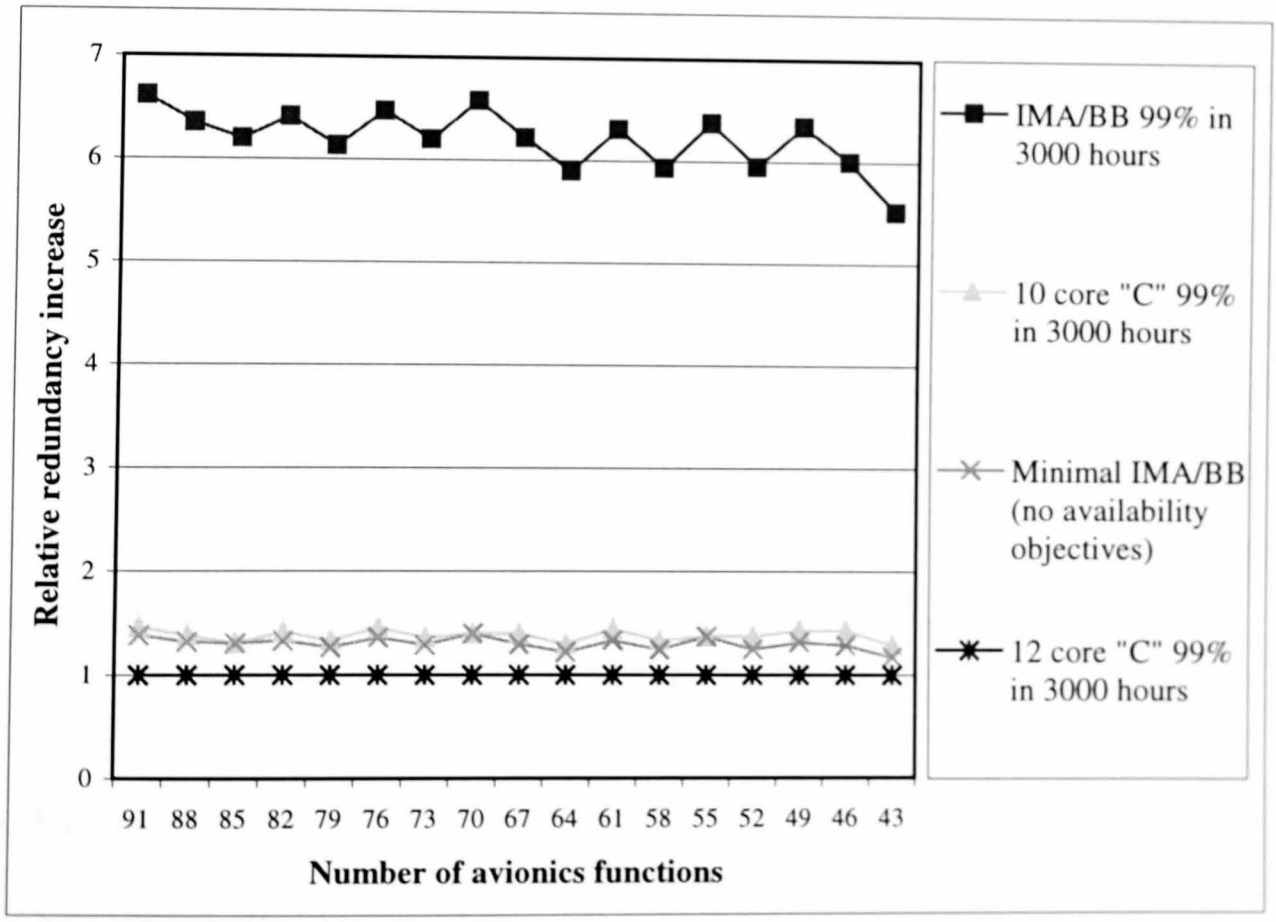



**Figure 11.2. Relative core LRM redundancy figures for "C-check" systems.**

The study of processing modules redundancy indicates that the use of reconfigurable systems should lead to significant savings when compared with non-reconfigurable systems. Moreover, as the IMA systems

should also lead to a decrease of the system weight, occupied space etc. [64], the combination of a reconfiguration scheme and an IMA architecture appears to constitute the next logical step. A detailed cost of ownership analysis would be required to properly assess all the benefits. However, as already mentioned, due to the sensitive nature of the data required to perform such a study the author was unable to complete this step.

## 11.3. Feasibility of RIMA

In order to conduct the feasibility analysis various autonomous reconfigurable systems have been implemented throughout the project. This involved a preliminary software model used to test several reconfiguration schemes, as well as two more representative implementations discussed in Chapter 9.

The study has shown that reconfigurable systems can perform well within their time constraints, should the hardware architecture be sufficiently reliable (see Chapter 9 for discussion). However, it has been observed that systems based on inadequate hardware or software platforms, e.g. a non-deterministic asynchronous backplane bus or inconsistent operating system, could pose a severe safety risk to an aircraft and as such should be subject to detailed certification procedures (these had been discussed in Chapter 10).

In general, the results obtained from the simulation of various configurations of RIMA are strongly encouraging, although they raise issues related to hardware and software acceptability. Should these issues be successfully resolved, the RIMA systems are expected to be able to meet their requirements, and should provide a cheaper and safer alternative to traditional Black Box based systems or IMA.

## 11.4. Recommendations for future work

Although the work completed within this research allowed for the initial implementation of a reconfigurable system, some aspects of RIMA should require additional consideration. These, due to the time constraints of the project, could not be completed, and they are be briefly discussed as recommendations for future work.

### 11.4.1. Power supply failure

It has been mentioned before that a failure of the power supply units could lead to conditions perceived by the system as simultaneous failures of multiple processing modules. Simultaneous events occurring in the system raise questions about the determinism of the reconfiguration scheme, that have been discussed in Chapter 8. Therefore, some additional work should be required to identify ways of providing reliable power sources for the cabinet, for example by employing replicated power supply modules with a probability of failure lower than the extremely improbable level, or by separating power sources related to particular core LRMs, such that a failure of a power supply unit does not affect multiple processing modules.

### 11.4.2. Application state

As some of the avionics applications require their state to be known on re-execution, the issue of automated extraction of the relevant state parameters and their communication to the application should be further investigated. The automated approach could allow relatively easy transition from a non-reconfigurable system to RIMA, without the need for a tedious analysis of individual avionics functions aimed at identifying their essential state variables.

Note that similar problems occur generally in distributed computing systems and are being researched (for example [23]), thus it can be expected that some of the existing solutions could be used or adapted to RIMA systems.

### 11.4.3. Data bus study

Although the ARINC 651 report suggests the use of the ARINC 659 data bus for a backplane, it has been previously discussed that the very static way in which the data bus access is constrained within the standard renders the dynamically reconfigurable approach extremely difficult to implement if not impossible. Therefore, an additional study should be performed leading to the identification of a desirable data bus standard and its implementation. Some initial work in this area is presented in [42], and it foresees the possibility of employing a modified ARINC 659 data bus in RIMA systems.

### 11.4.4. Fully representative implementation of RIMA

Mainly due to budget limitations neither of the implementations of RIMA constructed within this project was fully representative. In order to be able to properly analyse the behaviour of a reconfigurable system, its implementation should be based on representative hardware and software platforms, that should allow for detailed examination of the failure detection mechanism and other parts of the reconfiguration scheme.

The final system would employ a real-time operating system, appropriate application executive - conforming to the ARINC 653 standard - and adequate hardware architecture.

## 11.5. Conclusions

As RIMA is closely based on the principles of IMA, the benefits of reduced system weight, reduced number of different types of spare parts and others [64], [65] are also present in RIMA systems. However, the reconfigurable approach to avionics system design offers additional advantages related to its capacity for dynamic changes of the function to module assignment.

The Markov state space analysis of various RIMA systems has shown that the said systems are able to meet all the safety requirements and that they can satisfy very demanding dispatch availability objectives with greatly reduced redundancy of processing units, when compared with traditional systems. The configuration analysis has also shown that in the range of the most probable system sizes, the cabinet size of 10 or 12 core LRMs offers the greatest redundancy benefits allowing even for a 90% reduction of the number of redundant core LRMs.

Having established the most suitable architecture and its configuration, the work then focused on designing appropriate reconfiguration schemes. This led to the development of several different classes of reconfiguration algorithms based on common requirements and design methodology. As the aim of this phase was to develop a scheme to be implemented into an avionics system, methods employing function based static strategy tables were chosen, since they offer deterministic and predictable behaviour

controlled by an extremely simple algorithm. Such a scheme has subsequently been formally specified and its safety properties have been shown, which provided solid foundations for further implementation of the scheme during the demonstration of RIMA systems.

Despite the fact that both the software and hardware platforms used in the demonstration phase were not fully representative, the results of the simulation are very promising. It has to be noted at this point that although the more capable of the two systems was able to operate within its time constraints, the issue of inadequate hardware and software components had risen, and it is believed that future RIMA systems will have to address these issues in order to obtain the approval of the appropriate regulatory authorities. Various other certification issues were also identified and presented in Chapter 10.

Although there exist areas of RIMA that require further research, it is believed that RIMA systems are the natural step forward in the development of avionics systems, as they are able to provide greater safety and availability margins at an expected significantly reduced cost.

# Chapter 12. References

[1] Airlines Electronic Engineering Committee, 1991, *"Multi-Transmitter Databus"*, ARINC Report 629, AERONAUTICAL RADIO INC. (ARINC).

[2] Airlines Electronic Engineering Committee, 1991, *"Design Guidance for Integrated Modular Avionics"*, ARINC Report 651, AERONAUTICAL RADIO INC. (ARINC).

[3] JAR 25.1309, 1994 *"Joint Aviation Requirements, JAR-25 Large Aeroplanes"*, Change 14, 27 May 1994

[4] Karatshev P.S., Karatshev S.I., 1990, *"Supercomputer Architectures: Evolution and Implementations"*, in Supercomputing System, Van Nostrad Reinhold

[5] Apostolakis J., et al., 1990, *"Supercomputer Applications of the Hyper Cube"*, in Supercomputing System, Van Nostrad Reinhold

[6] Kozielski S., Szczerbinski Z., 1993, *"Parallel Computers: Architecture, Elements of Programming"* (in Polish), Wydawnictwo Naukowo-Techniczne

[7] Hockeny, R.W., Jesshope C.R., 1988, *"Parallel Computers 2: Architecture, Programming and Algorithms"*, IOP Publishing Ltd.

[8] Chalmers, A., 1994, *"Advanced Computer Architectures"*, Lecture Notes in the Department of Computer Science, University of Bristol

[9] Airlines Electronic Engineering Committee, 1993, *"Backplane Data Bus"*, ARINC Report 659, AERONAUTICAL RADIO INC. (ARINC).

[10] Airlines Electronic Engineering Committee, 1996, *"Avionics Application Software Standard Interface"*, Draft 15 of Project Paper 653, AERONAUTICAL RADIO INC. (ARINC).

[11] Mosse, D., Noh, S.H., Trinh, B., Agrawala, A.K., 1993, *"Multiple Resource Allocation for Multiprocessor Distributed Real-Time Systems"*, in IEEE Proceedings: Workshop on Parallel and Distributed Real-Time Systems.

[12] Yu, C., Das, C.R., 1995, *"Disjoint Task Allocation Algorithms for MIN Machines with Minimal Conflicts"*, in IEEE Transactions on Parallel and Distributed Systems.

[13] Woodside, C.M., Monforton, G.G., 1993, *"Fast Allocation of Process in Distributed and Parallel Systems"*, in IEEE Transactions on Parallel and Distributed Systems.

[14] Kim, J., Das, C.R., Lin, W., 1991, *"A Top-Down Processor Allocation Scheme for Hypercube Computers"*, in IEEE Transactions on Parallel and Distributed Systems.

[15] Stoica, I., Adel-Wahab, H., Jeffay, K., *"A Proportional Share resource Allocation Algorithm*

*for Real-Time, Time-Shared Systems"*.

[16] Mosse, D., *"Mechanisms for System-Level Fault Tolerance in Real-Time Systems"*, Department of Computer Science, University of Pittsburgh.

[17] Kao, B., Garcia-Molina, H., Adelberg B, *"On Building Distributed Real-Time Systems"*, Tech Report, University of Stanford.

[18] Artsy, Y., Finkel, R., 1989, *"Designing a Process Migration Facility: The Charlotte Experience"*, in IEEE Computer, '89.

[19] Powell, D., 1994, *"Distributed Fault-Tolerance - Lessons Learnt from Delta-4"* , in Hardware and Software Architectures for Fault-Tolerance - Experiments and Perspectives, Springer-Verlag, Berlin, Heidelberg.

[20] Dandamudi, S.P., Cheng, P.S.P., 1995, *"A Hierarchical Task Queue Organization for Shared Memory Multiprocessor System"*, in IEEE transactions on Parallel and Distributed Systems.

[21] Lo, V., 1988, *"Algorithms for Task Assignment in Distributed Systems"*, in IEEE Transactions on Computers, Nov. '88.

[22] Fernandez-Baca, D., Medepalli, A., *"Exact and Approximate Algorithms for Assignment Problems in Distributed Systems"*, Department of Computer Science and Department of Mathematics, Iowa State University.

[23] Hofmeister, C., Purtilo, J., *"A Framework for Dynamic Reconfiguration of Distributed Programs"*, University of Maryland.

[24] Liebeskind-Habas, R., Shrivastava, N., Melhem, R.G., Liu, C.L., 1995, *"Optimal Reconfiguration Algorithm for Real-Time Fault-Tolerant Processor Arrays"* , in IEEE Transactions on Parallel and Distributed Systems, May '95.

[25] Huang, Y., Kintala, C., Jalote, P., 1994, *"Two Techniques for Transient Software Error Recovery"*, in Hardware and Software Architectures for Fault-Tolerance - Experiments and Perspectives, Springer-Verlag, Berlin, Heidelberg.

[26] Lim, A., *"A State Machine Approach to Reliable and Dynamically Reconfigurable Distributed Systems"*, PhD Thesis, University of Wisconsin-Madison.

[27] Gosh, S., Melhem, R., Mosse, D., 1994, *"Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System"*, in International Parallel Processing Symposium.

[28] Mosse, D., Melhem, R., Ghosh, S., *"Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm"*, University of Pittsburgh.

[29] Ghosh, S., Melhem, R., Mosse, D., 1995, *"Enhancing Real-Time Schedules to Tolerate*

*Transient Faults*", in Real-Time Systems Symposium.

[30] Holland, G.D., Pradhan, D.K., "*A Software Implemented Fault-Tolerance Layer for Reliable Computing on Massively Parallel Computers and Distributed Computer Systems*", Department of Computer Science, Texas A&M University.

[31] Johnson, D., 1996, "*Integrated Modular Avionics - A Scheme for Autonomous Dynamic System Reconfiguration*", International Journal of Computer Systems Science and Engineering, May 1996, CRL Publishing Ltd.

[32] Omiecinski, T., 1996, "*Reconfigurable Integrated Modular Avionics – Implementation of Markov Analysis into Availability and Reliability Assessment of RIMA Systems*", Report 749, Department of Aerospace Engineering, University of Bristol.

[33] Blin, A., et al., 1978, "*Use of Markov Processes Reliability Problems*", Synthesis and Analysis Methods for Safety and Reliability Studies, London: Plenum Press.

[34] Somma, R., 1978, "*Some considerations on the Markov approach to reliability*", in Synthesis and Analysis Methods for Safety and Reliability Studies, London: Plenum Press.

[35] Clarotti, C.A., et al., 1978, "*Repairable Multiphase Systems - Markov and Fault Tree Approaches for Reliability Evaluation*", in Synthesis and Analysis Methods for Safety and Reliability Studies, London: Plenum Press.

[36] AMJ 25.1309 "*System Design and Analysis*", in Advisory Material Joint – AMJ

[36] ARP 4754, Systems Integration Requirements Task Group AS-1C, ASD, SAE, 1994, "*Guidelines for Certification of Highly-Integrated or Complex Aircraft Systems*".

[38] Rivest, R.L., 1992, "*RFC 1321: The MD5 Message-Digest Algorithm*", Internet Activities Board.

[39] Kesteloot, L., "*Fault-Tolerant Distributed Consensus*", Internet source, URL: http://tofu.alt.net/~lk/290.paper/290.paper.html

[40] Schneider F. B., 1993, "*What Good are Models and What Models are Good*", in Distributed Systems, ACM Press

[41] Clark R.N, et al, 1989, "*Fault diagnosis in dynamic systems : theory and applications*", Prentice Hall, New York

[42] Town S.A., 1998, "*Assessment of the ARINC 659 Data Bus for use within Reconfigurable Modular Avionics System*", Final year project report, Department of Aerospace Engineering, University of Bristol.

[43] Woodman, M. Heal, B., 1993, "*Introduction to VDM*", McGraw-Hill, London

[44] Andrews, D., Ince, D., 1991, *"Practical formal methods with VDM"*, McGraw-Hill, London.

[45] Jones, C.B., 1990, *"Systematic software development using VDM"*, 2nd ed., Prentice Hall, New York

[46] Liddel, D., 1994, *"Simple Design makes Reliable Computers"*, in Hardware and Software Architectures for Fault-Tolerance – Experiments and Perspectives, Springer-Verlag, Berlin, Heidelberg.

[47] The VDM-SL Tool Group, 1996, *"User Manual for the IFAD VDM-SL Toolbox"*, The Institute of Applied Computer Science, December 1996

[48] The VDM-SL Tool Group, 1996, *"The IFAD VDM-SL Language"*, The Institute of Applied Computer Science, December 1996

[49] 1996, *"Spec Box User Manual"*, Manual version PC/2.21a, 21st May, 1996, Adelard

[50] Jones, R.B., *"Methods and Tools for the Verification of Critical Properties"*, International Computer Limited, Internet source, URL: http://www.lemma-one.demon.co.uk/ProofPower/Papers/wrk050.ps.gz

[51] Lala, J.H. and Harper, R.E., 1994, *"Architectural Principles for Safety-Critical Real-Time Applications"*, Proceedings of the IEEE Vol. 82 No. 1 January 1994 (IEEE) pp 25-40.

[52] Metzger, P., 1996, *"Anatomia PC"*, in Polish, Wydawnictwo HELION

[53] Hamming, R.W., 1950, *"Error Detecting and Error Correcting Codes"*, Bell System Technical Journal 29, pp 147-160

[54] Jones, J. and Coghlan, B., 1994, *"Stable Disk – A Fault Tolerant Cached RAID Subsystem"*, in Hardware and Software Architectures for Fault-Tolerance - Experiments and Perspectives, Springer-Verlag, Berlin, Heidelberg.

[55] WindRiver Systems, 1997, *"VxWorks Tornado User's Guide 1.0"*, WindRiver Systems

[56] WindRiver Systems, 1995, *"VxWorks 5.3 Programmer's Guide"*, WindRiver Systems

[57] WindRiver Systems, 1995, *"VxWorks 5.3 Reference Manual"*, WindRiver Systems

[58] Rochkind, M.J., 1985, *"Advanced UNIX programming"*, Prentice-Hall

[59] Lapin, J.E., 1987, *"Portable C and UNIX system programming"*, Rabbit Software, Prentice-Hall

[60] *"IRIX™ Network Programming Guide"*, On-line books in IRIX™ 6.2 Operating System

[61] Mullender, S.J., 1993, *"Interprocess Communication"*, in Distributed Systems, ACM Press

[62] Hadzilacos V., Toueg S., 1993, *"Fault-Tolerant Broadcasts and Related Problems"*, in

Distributed Systems, ACM Press

[63] Veríssimo, P., 1993, *"Real-Time Communication"*, in Distributed Systems, ACM Press

[64] Johnson, D.M., Omiecinski, T.A., 1998, *"The feasibility and benefits of dynamic reconfiguration in integrated modular avionics"*, in The Aeronautical Journal, pp. 99 – 106, February 1998, The Royal Aeronautical Society

[65] Johnson, D., 1995, *"Autonomous Dynamic System Reconfiguration in Integrated Modular Avionics"*, A Research Proposal to the Civil Aviation Authority

# Appendix A. Examples of the assignment of software to core LRMs for the RIMA architecture "C" option without a software downloading bus.

Legend:

| | |
|---|---|
| hazardous | copies assigned to modules performing functions whose loss would lead to hazardous failure conditions |
| major | copies assigned to modules performing functions whose loss would lead to major failure conditions |
| minor | copies assigned to modules performing functions whose loss would lead to minor failure conditions |
| redundant | copies assigned to redundant modules |
| memory | module non-volatile on-board memory blocks (1 MB each) |
| $c_x$ | a copy of the x-th catastrophic function |
| $h_x$ | a copy of the x-th hazardous function |
| $maj_x$ (alternatively maj if only one function exists) | a copy of the x-th major function |
| $min_x$ (alternatively min if only one function exists) | a copy of the x-th minor function |
| option (x) for n core LRM cabinet | the x-th option as described in the tables above (Table 4.14, Table 4.15 and Table 4.16) for n core LRM cabinets |

Note that all assignments shown in this appendix were chosen to minimise the processing module memory requirements.

Appendix A. Examples of the assignment of software to core LRMs for the RIMA architecture "C" option without a software downloading bus.

## A.1. Assignment of software copies to processing modules - examples for 10 core LRM cabinets.

1.1. Option (a)

| Hazardous | | | Major | | Minor | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 2 | 1 | 1 | 2 | |
| c1 | c1 | c1 | h1 | h1 | maj1 | min | min | 1 MB |
| c2 | c2 | c2 | h2 | h2 | maj2 | maj1 | maj1 | 2 MB |
| | | | h3 | h3 | h1 | maj2 | maj2 | 3 MB |
| | | | c1 | c2 | h2 | h1 | h3 | 4 MB |
| | | | | | | h3 | h2 | 5 MB |

1.2. Option (b).

| Hazardous | | | Major | Minor | | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 1 | 2 | 1 | 2 | |
| c1 | c1 | c1 | h1 | h1 | maj | min1 | min1 | 1 MB |
| c2 | c2 | c2 | h2 | h2 | h1 | min2 | min2 | 2 MB |
| | | | h3 | h3 | h2 | maj | maj | 3 MB |
| | | | c1 | c2 | h3 | h1 | h3 | 4 MB |
| | | | | | | | h2 | 5 MB |

1.3. Option (c).

| Hazardous | | | Major | Minor | Redundant | | Memory |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 1 | 1 | 2 | |
| c1 | c1 | c1 | h1 | maj | min | min | 1 MB |
| c2 | c2 | c2 | h2 | h1 | maj | maj | 2 MB |
| c3 | c3 | c3 | h3 | h2 | h1 | h1 | 3 MB |
| | | | c1 | h3 | h2 | h2 | 4 MB |
| | | | c2 | c3 | h3 | h3 | 5 MB |

1.4. Option (d).

| Hazardous | | | | Major | Minor | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 1 | 1 | 1 | 2 | |
| c1 | c1 | c1 | c1 | h1 | h1 | min | min | 1 MB |
| c2 | c2 | c2 | c2 | h2 | h2 | maj | maj | 2 MB |
| | | | | h3 | h3 | h1 | h1 | 3 MB |
| | | | | h4 | h4 | h2 | h2 | 4 MB |
| | | | | | maj | h3 | h3 | 5 MB |
| | | | | | | h4 | h4 | 6 MB |

1.5. Option (e).

| Hazardous | | | | | Major | Minor | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 1 | 1 | 1 | 2 | |
| c1 | c1 | c1 | c1 | | h1 | h1 | min | min | 1 MB |
| | | | | | h2 | h2 | maj | maj | 2 MB |
| | | | | | h3 | h3 | h1 | h1 | 3 MB |
| | | | | | h4 | h4 | h2 | h2 | 4 MB |
| | | | | | h5 | h5 | h3 | h3 | 5 MB |
| | | | | | | maj | h4 | h4 | 6 MB |
| | | | | | | | h5 | h5 | 7 MB |

## A.2. Assignment of software copies to processing modules - examples for 12 core LRM cabinets.

2.1. Option (a).

| Hazardous | | | Major | | | Minor | | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | |
| c1 | c1 | c1 | c1 | c2 | h1 | h1 | maj1 | min1 | min1 | 1 MB |
| c2 | c2 | c2 | h1 | h1 | h2 | h2 | maj2 | min2 | min2 | 2 MB |
| | | | h2 | h2 | h3 | h3 | maj3 | maj1 | maj1 | 3 MB |
| | | | h3 | h3 | | | | maj2 | maj2 | 4 MB |
| | | | | | | | | maj3 | maj3 | 5 MB |

2.2. Option (b).

| Hazardous | | | | Major | | Minor | | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 1 | 2 | 1 | 2 | 1 | 2 | |
| c1 | c1 | c1 | c1 | h1 | h1 | maj1 | maj2 | min1 | min1 | 1 MB |
| c2 | c2 | c2 | c2 | h2 | h2 | h1 | h1 | min2 | min2 | 2 MB |
| | | | | h3 | h3 | h2 | h2 | maj1 | maj1 | 3 MB |
| | | | | h4 | h4 | h3 | h3 | maj2 | maj2 | 4 MB |
| | | | | | | | | h4 | h4 | 5 MB |

2.3. Option (c).

| Hazardous | | | | Major | | Minor | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 1 | 2 | 1 | 1 | 2 | |
| c1 | c1 | c1 | c1 | h1 | h1 | maj1 | min | min | 1 MB |
| c2 | c2 | c2 | c2 | h2 | h2 | maj2 | maj1 | maj1 | 2 MB |
| c3 | c3 | c3 | c3 | h3 | h3 | h1 | maj2 | maj2 | 3 MB |
| | | | | h4 | h4 | h2 | h1 | h2 | 4 MB |
| | | | | | | h3 | h3 | h4 | 5 MB |
| | | | | | | | h4 | | 6 MB |

2.4. Option (d).

| Hazardous | | | | Major | Minor | | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 1 | 1 | 2 | 1 | 2 | |
| c1 | c1 | c1 | c1 | h1 | h1 | h1 | min1 | min1 | 1 MB |
| c2 | c2 | c2 | c2 | h2 | h2 | h2 | min2 | min2 | 2 MB |
| c3 | c3 | c3 | c3 | h3 | h3 | h3 | maj | maj | 3 MB |
| | | | | h4 | h4 | h4 | h1 | h3 | 4 MB |
| | | | | | | maj | h2 | h4 | 5 MB |

2.5. Option (e).

| Hazardous | | | | | Major | Minor | Redundant | | Memory |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 1 | 1 | 1 | 2 | |
| c1 | c1 | c1 | c1 | | h1 | h1 | h1 | h1 | 1 MB |
| c2 | c2 | c2 | c2 | | h2 | h2 | h2 | h2 | 2 MB |
| c3 | c3 | c3 | c3 | | h3 | h3 | h3 | h3 | 3 MB |
| | | | | | h4 | h4 | h4 | h4 | 4 MB |
| | | | | | h5 | h5 | h5 | h5 | 5 MB |
| | | | | | | maj | maj | maj | 6 MB |
| | | | | | | | min | min | 7 MB |

## A.3. Assignment of software copies to processing modules - examples for 16 core LRM cabinets.

3.1. Option (a).

| Hazardous | | | | | Major | | | Minor | | Redundant | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 3 | |
| c1 | c1 | c1 | c1 | c1 | h1 | h1 | h1 | maj1 | maj1 | min1 | min1 | min1 | 1 MB |
| c2 | c2 | c2 | c2 | c2 | h2 | h2 | h2 | maj2 | maj2 | min2 | min2 | min2 | 2 MB |
| c3 | c3 | c3 | c3 | c3 | h3 | h3 | h3 | maj3 | maj3 | maj1 | maj1 | maj2 | 3 MB |
| | | | | | h4 | h4 | h4 | h1 | h1 | maj2 | maj3 | maj3 | 4 MB |
| | | | | | h5 | h5 | h5 | h2 | h2 | h3 | h3 | h4 | 5 MB |
| | | | | | | | | | | h4 | h5 | h5 | 6 MB |

3.2. Option (b).

| Hazardous | | | | | Major | | Minor | | Redundant | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 1 | 2 | 1 | 2 | 1 | 2 | 3 | |
| c1 | c1 | c1 | c1 | c1 | h1 | h1 | h1 | maj1 | maj1 | maj1 | maj1 | 1 MB |
| c2 | c2 | c2 | c2 | c2 | h2 | h2 | h2 | maj2 | maj2 | maj2 | maj2 | 2 MB |
| c3 | c3 | c3 | c3 | c3 | h3 | h3 | h3 | h3 | min1 | min1 | min1 | 3 MB |
| c4 | c4 | c4 | c4 | c4 | h4 | h4 | h4 | h4 | min2 | min2 | min2 | 4 MB |
| | | | | | h5 | h5 | h5 | h5 | h1 | h1 | h3 | 5 MB |
| | | | | | | | | | h2 | h2 | h4 | 6 MB |
| | | | | | | | | | | | h5 | 7 MB |

3.3. Option (c).

| Hazardous | | | | Major | | | Minor | | | Redundant | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | |
| c1 | c1 | c1 | c1 | c1 | h1 | h1 | h1 | h1 | maj1 | min1 | min1 | min1 | 1 MB |
| c2 | c2 | c2 | c2 | c2 | h2 | h2 | h2 | h2 | maj2 | min2 | min2 | min2 | 2 MB |
| c3 | c3 | c3 | c3 | c3 | h3 | h3 | h3 | h3 | h2 | min3 | min3 | min3 | 3 MB |
| | | | | h1 | h4 | h4 | h4 | h4 | h3 | maj1 | maj1 | maj1 | 4 MB |
| | | | | | | | | maj3 | h4 | maj2 | maj2 | maj2 | 5 MB |
| | | | | | | | | | | maj3 | maj3 | maj3 | 6 MB |

3.4. Option (d).

| Hazardous | | | | | | Major | Minor | Redundant | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 1 | 1 | 1 | 2 | 3 | |
| c1 | c1 | c1 | c1 | c1 | | h1 | maj | maj | maj | maj | 1 MB |
| c2 | c2 | c2 | c2 | c2 | | h2 | h1 | min | min | min | 2 MB |
| c3 | c3 | c3 | c3 | c3 | | h3 | h2 | h1 | h1 | h1 | 3 MB |
| c4 | c4 | c4 | c4 | c4 | | h4 | h3 | h2 | h2 | h2 | 4 MB |
| c5 | c5 | c5 | c5 | c5 | | h5 | h4 | h3 | h3 | h3 | 5 MB |
| | | | | | | h6 | h5 | h4 | h4 | h4 | 6 MB |
| | | | | | | | h6 | h5 | h5 | h5 | 7 MB |
| | | | | | | | | h6 | h6 | h6 | 8 MB |

3.5. Option (e).

| Hazardous | | | | | | | Major | Minor | Redundant | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 1 | 1 | 2 | 3 | |
| c1 | c1 | c1 | c1 | c1 | | | h1 | maj | maj | maj | maj | 1 MB |
| c2 | c2 | c2 | c2 | c2 | | | h2 | h1 | min | min | min | 2 MB |
| c3 | c3 | c3 | c3 | c3 | | | h3 | h2 | h1 | h1 | h1 | 3 MB |
| c4 | c4 | c4 | c4 | c4 | | | h4 | h3 | h2 | h2 | h2 | 4 MB |
| | | | | | | | h5 | h4 | h3 | h3 | h3 | 5 MB |
| | | | | | | | h6 | h5 | h4 | h4 | h4 | 6 MB |
| | | | | | | | h7 | h6 | h5 | h5 | h5 | 7 MB |
| | | | | | | | | h7 | h6 | h6 | h6 | 8 MB |
| | | | | | | | | | h7 | h7 | h7 | 9 MB |